

## Dal catalogo Apogeo

### *Informatica*

Biermann, Ramm, *Le idee dell'informatica*

Bolchini, Brandolesi, Salice, Sciuto, *Reti logiche*, seconda edizione

Coppola, Mizzaro, *Laboratorio di programmazione in Java*

Bruni, Corradini, Gervasi, *Programmazione in Java*

Deitel, *C Corso completo di programmazione*, terza edizione

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

Hennessy, Patterson, *Architettura degli elaboratori*

King, *Programmazione in C*

Laganà, Righi, Romani, *Informatica. Concetti e sperimentazioni*, seconda edizione

Lombardo, Valle, *Audio e multimedia*, terza edizione

Mazzanti, Milanese, *Programmazione di applicazioni grafiche in Java*

Peterson, Davie, *Reti di calcolatori*, seconda edizione

Pigni, Ravarini, Sciuto, *Sistemi per la gestione dell'informazione*, seconda edizione

Polillo, *Facile da usare. Una moderna introduzione all'ingegneria dell'usabilità*

Schneider, Gersting, *Informatica*

# Concetti di informatica e fondamenti di Java

quinta edizione

Cay Horstmann

Edizione italiana a cura di  
Marcello Dalpasso

**APGEO**

# **Concetti di informatica e fondamenti di Java**

## **Quinta edizione**

Titolo originale:  
**Java Concepts, 6<sup>th</sup> edition**

Autore:  
**Cay Horstmann**

Copyright © 2010 **John Wiley & Sons**, Inc. All rights reserved.  
Authorized translation from the English language edition published by John Wiley & Sons, Inc.

Copyright © 2010 – **APOGEO s.r.l.**  
Socio unico Giangiacomo Feltrinelli Editore s.r.l.  
Via Natale Battaglia, 12 – 20127 Milano (Italy)  
Telefono: 02-289981 (5 linee r.a.) – Telefax: 02-26116334  
Email [education@apogeonline.com](mailto:education@apogeonline.com)  
U.R.L. <http://www.apogeonline.com>

**ISBN 978-88-503-2956-4**

**Traduzione e revisione: Marcello Dalpasso**  
**Impaginazione elettronica: Grafica Editoriale – Vimercate**  
**Editor: Alberto Kratter Thaler**  
**Redazione: Patrizia Villani**  
**Copertina e progetto grafico: Enrico Marcandalli**

Tutti i diritti sono riservati a norma di legge e a norma delle convenzioni internazionali.  
Nessuna parte di questo libro può essere riprodotta con sistemi elettronici, meccanici o altri,  
senza l'autorizzazione scritta dell'Editore.

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

Fotocopie per uso personale del lettore possono essere effettuate nei limiti del 15% di ciascun volume dietro pagamento alla SIAE del compenso previsto dall'art. 68, comma 4, della legge 22 aprile 1941 n. 633 ovvero dell'accordo stipulato tra SIAE, AIE, SNS e CNA, CONFARTIGIANATO, CASA, CLAAI, CONFCOMMERCIO, CONFESERCENTI il 18 dicembre 2000.

Le riproduzioni a uso differente da quello personale potranno avvenire, per un numero di pagine non superiore al 15% del presente volume, solo a seguito di specifica autorizzazione rilasciata da AIDRO, C.so di Porta Romana, n. 108, 20122 Milano, telefono 02 89280804, telefax 02 892864, e-mail [aidro@iol.it](mailto:aidro@iol.it).

Finito di stampare nel mese di settembre 2010  
da L.e.g.o. – Lavis, Trento

# Sommario

<b>Sommario delle sezioni speciali .....</b>	<b>xiii</b>
<b>Presentazione della quinta edizione italiana .....</b>	<b>xix</b>
<b>Prefazione .....</b>	<b>xxi</b>
<b>Capitolo 1 – Introduzione .....</b>	<b>1</b>
Obiettivi del capitolo .....	1
1.1 Che cos’è la programmazione? .....	2
1.2 L’anatomia di un computer .....	3
1.3 Passare da programmi leggibili al codice macchina .....	6
1.4 Il linguaggio di programmazione Java .....	8
1.5 La struttura di un semplice programma .....	11
1.6 Compilare ed eseguire un programma Java .....	15
1.7 Errori .....	19
1.8 Algoritmi .....	21
Riepilogo degli obiettivi di apprendimento .....	29
Classi, oggetti e metodi presentati nel capitolo .....	30
Esercizi di ripasso .....	30
<b>Capitolo 2 – Utilizzare gli oggetti .....</b>	<b>33</b>
Obiettivi del capitolo .....	33
2.1 Tipi .....	34

2.2 Variabili .....	35
2.3 L'operatore di assegnazione .....	38
2.4 Oggetti, classi e metodi .....	41
2.5 Parametri e valori restituiti dei metodi .....	43
2.6 Costruire oggetti .....	46
2.7 Metodi d'accesso e metodi modificatori .....	48
2.8 La documentazione API .....	49
2.9 Realizzare un programma di collaudo .....	53
2.10 Riferimenti a oggetti .....	62
2.11 Applicazioni grafiche e finestre .....	64
2.12 Disegnare all'interno di un componente .....	67
2.13 Ellissi, linee, testo e colore .....	73
2.13.1 Ellissi e cerchi .....	74
2.13.2 Linee .....	75
2.13.3 Scrivere testo .....	75
2.13.4 Colori .....	76
Riepilogo degli obiettivi di apprendimento .....	79
Classi, oggetti e metodi presentati nel capitolo .....	81
Esercizi di ripasso .....	81
<b>Capitolo 3 – Realizzare classi.....</b>	<b>83</b>
Obiettivi del capitolo .....	83
3.1 Variabili di esemplare .....	84
3.2 Incapsulamento .....	86
3.3 Progettare l'interfaccia pubblica di una classe .....	88
3.4 Commentare l'interfaccia pubblica .....	91
3.5 Realizzare la classe .....	95
3.6 Collaudo di unità .....	106
3.7 Variabili locali .....	108
3.8 Parametri impliciti .....	110
3.9 Classi per figure complesse .....	113
Riepilogo degli obiettivi di apprendimento .....	123
Esercizi di ripasso .....	124
<b>Capitolo 4 – Tipi di dati fondamentali.....</b>	<b>127</b>
Obiettivi del capitolo .....	127
4.1 Tipi di numeri .....	128
4.2 Costanti .....	132
4.3 Operazioni aritmetiche e funzioni matematiche .....	138
4.3.1 Operatori aritmetici .....	138
4.3.2 Incremento e decremento .....	139
4.3.3 Divisione intera .....	139
4.3.4 Potenze e radici .....	140
4.3.5 Conversione e arrotondamento .....	142
4.4 Invocare metodi statici .....	147
4.5 Stringhe .....	155
4.5.1 La classe <code>String</code> .....	155

4.5.2	Concatenazione.....	156
4.5.3	Convertire stringhe in numeri .....	156
4.5.4	Sottostringhe.....	157
4.6	Leggere dati in ingresso .....	161
	Riepilogo degli obiettivi di apprendimento .....	169
	Classi, oggetti e metodi presentati nel capitolo .....	169
	Esercizi di ripasso .....	170
<b>Capitolo 5 – Decisioni .....</b>		<b>173</b>
	Obiettivi del capitolo .....	173
5.1	L'enunciato <code>if</code> .....	174
5.2	Confrontare valori .....	179
5.2.1	Operatori relazionali.....	179
5.2.2	Confrontare numeri in virgola mobile .....	180
5.2.3	Confrontare stringhe .....	181
5.2.4	Confrontare oggetti .....	184
5.2.5	Confrontare con <code>null</code> .....	185
5.3	Alternative multiple .....	191
5.3.1	Sequenze di confronti.....	191
5.3.2	Diramazioni annidate.....	194
5.4	Utilizzare espressioni booleane .....	203
5.4.1	Il tipo <code>boolean</code> .....	203
5.4.2	I metodi predicativi .....	203
5.4.3	Gli operatori booleani .....	204
5.4.4	Utilizzare variabili booleane.....	205
5.5	Collaudo e copertura del codice.....	209
	Riepilogo degli obiettivi di apprendimento .....	215
	Classi, oggetti e metodi presentati nel capitolo .....	215
	Esercizi di ripasso .....	216
<b>Capitolo 6 – Iterazioni .....</b>		<b>221</b>
	Obiettivi del capitolo .....	221
6.1	Cicli <code>while</code> .....	222
6.2	Cicli <code>for</code> .....	231
6.3	Algoritmi comuni che usano cicli.....	241
6.3.1	Calcolo di un totale .....	241
6.3.2	Conteggio di eventi.....	242
6.3.3	Identificazione della prima corrispondenza .....	242
6.3.4	Richiesta ripetuta fino al raggiungimento di un obiettivo .....	242
6.3.5	Confronto di valori adiacenti.....	243
6.3.6	Lettura di dati mediante valori “sentinella” .....	244
6.4	Cicli annidati .....	256
6.5	Numeri casuali e simulazioni.....	260
6.6	Usare un debugger .....	265
	Riepilogo degli obiettivi di apprendimento .....	277
	Classi, oggetti e metodi presentati nel capitolo .....	278
	Esercizi di ripasso .....	278

<b>Capitolo 7 – Vettori e array.....</b>	<b>281</b>
Obiettivi del capitolo .....	281
7.1 Array .....	282
7.2 Vettori (liste ad accesso casuale) .....	290
7.3 Classi involucro ( <i>wrapper</i> ) e auto-boxing .....	296
7.4 Il ciclo <code>for</code> esteso .....	298
7.5 Array riempiti solo in parte .....	300
7.6 Semplici algoritmi per vettori e array .....	302
7.6.1 Riempimento.....	302
7.6.2 Calcolare somme e valori medi.....	303
7.6.3 Contare valori aventi determinate caratteristiche.....	303
7.6.4 Trovare il valore massimo o minimo .....	303
7.6.5 Trovare un valore.....	304
7.6.6 Trovare la posizione di un elemento.....	304
7.6.7 Eliminare un elemento .....	305
7.6.8 Inserire un elemento .....	306
7.6.9 Copiare e ingrandire array .....	307
7.6.10 Visualizzare gli elementi con separatori.....	308
7.7 Collaudo regressivo.....	320
7.8 Array a due dimensioni .....	323
Riepilogo degli obiettivi di apprendimento .....	331
Classi, oggetti e metodi presentati nel capitolo .....	332
Esercizi di ripasso .....	333
<b>Capitolo 8 – Progettazione di classi .....</b>	<b>337</b>
Obiettivi del capitolo .....	337
8.1 Scegliere le classi .....	338
8.2 Coesione e accoppiamento .....	339
8.3 Classi immutabili.....	342
8.4 Effetti collaterali.....	343
8.5 Pre-condizioni e post-condizioni .....	349
8.6 Metodi statici.....	354
8.7 Variabili statiche.....	357
8.8 Ambito di visibilità .....	361
8.8.1 Visibilità di variabili.....	361
8.8.2 Visibilità sovrapposte .....	362
8.9 Pacchetti .....	365
8.9.1 Organizzare classi in pacchetti .....	366
8.9.2 Importare pacchetti .....	366
8.9.3 Nomi di pacchetto .....	367
8.9.4 Come vengono localizzate le classi.....	368
8.10 Ambienti per il collaudo di unità.....	373
Riepilogo degli obiettivi di apprendimento .....	375
Esercizi di ripasso .....	376

<b>Capitolo 9 – Interfacce e polimorfismo .....</b>	<b>381</b>
Obiettivi del capitolo .....	381
9.1 Uso di interfacce per il riutilizzo del codice .....	382
9.2 Conversione di tipo fra classe e interfaccia .....	388
9.3 Polimorfismo .....	390
9.4 Usare interfacce di smistamento ( <i>callback</i> ) .....	393
9.5 Classi interne .....	398
9.6 Oggetti semplificati .....	401
9.7 Eventi: ricevitori e sorgenti .....	404
9.8 Classi interne come ricevitori di eventi .....	407
9.9 Costruire applicazioni dotate di pulsanti .....	410
9.10 Elaborare eventi di temporizzazione .....	415
9.11 Eventi del mouse .....	418
Riepilogo degli obiettivi di apprendimento .....	424
Classi, oggetti e metodi presentati nel capitolo .....	425
Esercizi di ripasso .....	425
<b>Capitolo 10 – Ereditarietà .....</b>	<b>429</b>
Obiettivi del capitolo .....	429
10.1 Gerarchie di ereditarietà .....	430
10.2 Realizzare sottoclassi .....	432
10.3 Sovrascrivere metodi .....	437
10.4 Costruttori in sottoclassi .....	441
10.5 Conversione di tipo fra sottoclasse e superclasse .....	444
10.6 Polimorfismo e ereditarietà .....	446
10.7 La superclasse universale <code>Object</code> .....	462
10.7.1 Sovrascrivere il metodo <code>toString</code> .....	462
10.7.2 Sovrascrivere il metodo <code>equals</code> .....	464
10.7.3 Il metodo <code>clone</code> .....	465
10.8 L'ereditarietà per personalizzare i frame .....	473
Riepilogo degli obiettivi di apprendimento .....	476
Classi, oggetti e metodi presentati nel capitolo .....	477
Esercizi di ripasso .....	477
<b>Capitolo 11 – Ingresso/uscita e gestione delle eccezioni.....</b>	<b>481</b>
Obiettivi del capitolo .....	481
11.1 Leggere e scrivere file di testo .....	482
11.2 Acquisire testi .....	487
11.2.1 Leggere parole .....	488
11.2.2 Elaborare righe .....	489
11.2.3 Leggere numeri .....	490
11.2.4 Leggere caratteri .....	492
11.3 Lanciare eccezioni .....	501
11.4 Eccezioni controllate e non controllate .....	504
11.5 Catturare eccezioni .....	506
11.6 La clausola <code>finally</code> .....	509
11.7 Progettare eccezioni .....	512

11.8 Un esempio completo.....	513
Riepilogo degli obiettivi di apprendimento .....	519
Classi, oggetti e metodi presentati nel capitolo .....	520
Esercizi di ripasso.....	520
<b>Capitolo 12 – Ricorsione .....</b>	<b>523</b>
Obiettivi del capitolo .....	523
12.1 Numeri triangolari.....	524
12.2 Metodi ausiliari ricorsivi .....	538
12.3 L'efficienza della ricorsione .....	539
12.4 Permutazioni .....	545
12.5 Ricorsione mutua .....	549
Riepilogo degli obiettivi di apprendimento .....	558
Esercizi di ripasso.....	558
<b>Capitolo 13 – Ordinamento e ricerca.....</b>	<b>561</b>
Obiettivi del capitolo .....	561
13.1 Ordinamento per selezione .....	562
13.2 Misurazione delle prestazioni dell'ordinamento per selezione.....	565
13.3 Analisi delle prestazioni dell'algoritmo di ordinamento per selezione .....	569
13.4 Ordinamento per fusione (MergeSort) .....	573
13.5 Analisi dell'algoritmo di ordinamento per fusione.....	576
13.6 Effettuare ricerche.....	581
13.7 Ricerca binaria .....	584
13.8 Ordinare dati veri .....	587
Riepilogo degli obiettivi di apprendimento .....	591
Classi, oggetti e metodi presentati nel capitolo .....	592
Esercizi di ripasso.....	592
<b>Capitolo 14 – Introduzione alle strutture di dati .....</b>	<b>595</b>
Obiettivi del capitolo .....	595
14.1 Utilizzare liste concatenate .....	596
14.2 Realizzare liste concatenate .....	602
14.3 Tipi di dati astratti.....	614
14.4 Pile e code .....	619
Riepilogo degli obiettivi di apprendimento .....	624
Classi, oggetti e metodi presentati nel capitolo .....	625
Esercizi di ripasso.....	625
<b>Capitolo 15 – Strutture di dati avanzate .....</b>	<b>629</b>
Obiettivi del capitolo .....	629
15.1 Insiemi.....	630
15.2 Mappe .....	634
15.3 Tabelle hash .....	642
15.4 Calcolare codici hash .....	649
15.5 Alberi di ricerca binari.....	654
15.6 Visita di un albero .....	665

15.7	Code prioritarie.....	668
15.8	Heap.....	669
15.9	L'algoritmo <i>Heapsort</i> .....	680
	Riepilogo degli obiettivi di apprendimento .....	686
	Classi, oggetti e metodi presentati nel capitolo .....	687
	Esercizi di ripasso .....	688
<b>Capitolo 16 – Programmazione generica (disponibile online)</b>		
	Obiettivi del capitolo .....	W-1
16.1	Classi generiche e tipi parametrici.....	W-2
16.2	Realizzare tipi generici .....	W-3
16.3	Metodi generici .....	W-7
16.4	Vincolare i tipi parametrici.....	W-8
16.5	Cancellazione dei tipi ( <i>type erasure</i> ) .....	W-11
	Riepilogo degli obiettivi di apprendimento .....	W-14
	Esercizi di ripasso .....	W-15
<b>Esercizi e progetti di programmazione (disponibile online)</b>		
<b>Appendice A – Risposte alle domande di auto-valutazione .....</b>		691
<b>Appendice B – Linguaggio Java: operatori .....</b>		701
<b>Appendice C – Linguaggio Java: parole riservate .....</b>		703
<b>Appendice D – Sistemi di numerazione.....</b>		705
<b>Appendice E – Operazioni con bit e scorimenti.....</b>		711
<b>Appendice F – Il sottoinsieme Basic Latin di Unicode.....</b>		715
<b>Appendice G – Linguaggio Java: linee guida (disponibile online)</b>		
<b>Appendice H – Linguaggio Java: compendio sintattico (disponibile online)</b>		
<b>Glossario .....</b>		717
<b>Indice analitico.....</b>		729



# Sommario delle sezioni speciali



## Argomenti avanzati

1.1	Sintassi alternativa per i commenti .....	14
2.1	Collaudare classi in un ambiente interattivo.....	54
2.2	Applet.....	71
3.1	Invocare un costruttore dall'interno di un altro costruttore .....	112
4.1	Numeri grandi .....	130
4.2	Numeri binari .....	130
4.3	Combinare assegnazioni e operatori aritmetici .....	146
4.4	Sequenze di escape .....	159
4.5	Le stringhe e il tipo <code>char</code> .....	160
4.6	Visualizzare numeri nel formato desiderato.....	166
4.7	Ricezione e visualizzazione di dati con una finestra di dialogo.....	168
5.1	L'operatore di selezione o condizionale .....	179
5.2	L'enunciato <code>switch</code> .....	193
5.3	Tipi enumerativi .....	201
5.4	Valutazione "pigra" degli operatori booleani .....	208
5.5	La legge di De Morgan .....	208
5.6	Tracciamento .....	213
6.1	Cicli <code>do</code> .....	230
6.2	Visibilità delle variabili dichiarate nell'intestazione di un ciclo <code>for</code> .....	239
6.3	Il problema del "ciclo e mezzo".....	253

6.4	Gli enunciati <code>break</code> e <code>continue</code> .....	255
6.5	Condizione invariante in un ciclo .....	263
7.1	Metodi con un numero di parametri variabile .....	288
7.2	Miglioramenti per la sintassi di <code>ArrayList</code> in Java 7.....	295
7.3	Array bidimensionali con righe di lunghezze variabili.....	330
7.4	Array multidimensionali .....	331
8.1	Invocazione per valore e invocazione per riferimento.....	348
8.2	Invarianti di classe .....	353
8.3	Importazione statica.....	360
8.4	Forme alternative per inizializzare variabili di esemplare e variabili statiche.....	360
8.5	Accesso di pacchetto .....	369
9.1	Costanti nelle interfacce.....	387
9.2	Classi interne anonime.....	400
9.3	Adattatori per eventi .....	421
10.1	Classi astratte .....	448
10.2	Metodi e classi <code>final</code> .....	449
10.3	Accesso protetto.....	451
10.4	L'ereditarietà e il metodo <code>toString</code> .....	466
10.5	L'ereditarietà e il metodo <code>equals</code> .....	468
10.6	Realizzare il metodo <code>clone</code> .....	470
10.7	Una rivisitazione dei tipi enumerativi.....	472
10.8	Aggiungere il metodo <code>main</code> alla classe del frame.....	476
11.1	Finestre per la selezione di file.....	485
11.2	Leggere pagine web .....	486
11.3	Argomenti sulla riga dei comandi .....	486
11.4	Gestione automatica delle risorse in Java 7 .....	512
13.1	Ordinamento per inserimento.....	571
13.2	O-grande, Omega ( $\Omega$ ) e Theta ( $\Theta$ ) .....	572
13.3	L'algoritmo Quicksort.....	579
13.4	L'interfaccia <code>Comparable</code> parametrica.....	590
13.5	L'interfaccia <code>Comparator</code> .....	590
14.1	L'interfaccia <code>Iterable</code> e il ciclo <code>for</code> esteso .....	601
14.2	Classi interne statiche.....	614
15.1	Miglioramenti alle classi contenitore in Java 7 .....	637
16.1	Tipi con carattere jolly ( <i>wildcard</i> ).....	W–10



## Consigli per la produttività

1.1	Conoscere il <i>File System</i> .....	17
1.2	Adottare una strategia di <i>backup</i> .....	18
2.1	Non usate la memoria: usate la documentazione! .....	52
3.1	Il programma di utilità <code>javadoc</code> .....	94
4.1	Leggere le segnalazioni di eccezioni .....	158
5.1	Spostamenti verso destra e tabulazioni nel codice .....	177
5.2	Eseguire a mano e tenere traccia su carta .....	199
5.3	Preparare un piano e riservarsi tempo .....	200
6.1	Tenere traccia dell'esecuzione di cicli.....	226

7.1	Visualizzare array e vettori con facilità .....	312
7.2	File <i>batch</i> e <i>script</i> di <i>shell</i> .....	322
9.1	Non usate un contenitore come ricevitore di eventi .....	413
11.1	Espressioni canoniche ( <i>regular expressions</i> ) .....	492



## Consigli per la qualità

2.1	Scegliete nomi significativi per le variabili .....	38
4.1	Non usate numeri magici .....	138
4.2	Spaziature .....	144
4.3	Identificare nel codice le componenti comuni .....	145
5.1	Disposizione delle parentesi graffe .....	176
5.2	Evitare verifiche di condizioni che abbiano effetti collaterali .....	185
5.3	Calcolare manualmente dati di prova .....	212
5.4	Preparare in anticipo i casi di prova .....	212
6.1	Usare i cicli <b>for</b> solamente per lo scopo previsto .....	236
6.2	Non usare l'operatore <b>!=</b> per verificare la fine di un intervallo .....	238
6.3	Limiti simmetrici e asimmetrici .....	240
6.4	Contare le iterazioni .....	240
7.1	Usate array per sequenze di valori correlati .....	286
7.2	Trasformare array paralleli in array di oggetti .....	287
8.1	Coerenza .....	341
8.2	Ridurre al minimo gli effetti collaterali .....	346
8.3	Non modificate il contenuto di variabili parametro .....	347
8.4	Limitare al massimo l'uso di metodi statici .....	356
8.5	Minimizzare l'ambito di visibilità di ciascuna variabile .....	364
10.1	Realizzate il metodo <b>toString</b> in tutte le classi .....	466
10.2	Nei metodi d'accesso, clonate le variabili di esemplare modificabili .....	469
11.1	Lanciare presto, catturare tardi .....	508
11.2	Non mettete a tacere le eccezioni .....	509
11.3	Non usate <b>catch</b> e <b>finally</b> nel medesimo blocco <b>try</b> .....	511
11.4	Lanciate eccezioni veramente specifiche .....	513
15.1	Usare riferimenti a interfaccia per manipolare strutture dati .....	634



## Consigli pratici

1.1	Sviluppo e descrizione di un algoritmo .....	25
3.1	Realizzare una classe .....	99
3.2	Disegnare forme grafiche .....	118
4.1	Effettuare calcoli .....	148
5.1	Progettare un enunciato <b>if</b> .....	186
6.1	Realizzare cicli .....	247
6.2	L'attività di debugging .....	269
7.1	Usare array e vettori .....	312
8.1	Programmare con i pacchetti .....	370
10.1	Progettare una gerarchia di ereditarietà .....	452

11.1	Elaborare file di testo .....	493
12.1	Pensare ricorsivamente .....	528
15.1	Scegliere un contenitore .....	638



## Errori comuni

1.1	Dimenticare i punti e virgola .....	14
1.2	Errori di ortografia .....	21
2.1	Fare confusione tra gli enunciati di dichiarazione di variabile e di assegnazione.....	40
2.2	Cercare di invocare un costruttore come se fosse un metodo .....	48
3.1	Dichiarare un costruttore <code>void</code> .....	91
3.2	Dimenticarsi di inizializzare i riferimenti agli oggetti in un costruttore .....	109
4.1	Divisione fra numeri interi.....	143
4.2	Parentesi non accoppiate .....	144
4.3	Errori di arrotondamento.....	146
5.1	Un punto e virgola dopo la condizione dell' <code>if</code> .....	178
5.2	Utilizzare <code>==</code> per confrontare stringhe .....	182
5.3	Il problema dell' <code>else</code> sospeso .....	197
5.4	Espressioni con più operatori relazionali.....	206
5.5	Confondere le condizioni <code>&amp;&amp;</code> e <code>  </code> .....	207
6.1	Cicli infiniti.....	228
6.2	Errori per scarto di uno .....	228
6.3	Dimenticare un punto e virgola .....	237
6.4	Un punto e virgola di troppo .....	238
7.1	Errori di limiti .....	285
7.2	Array non inizializzati o non riempiti .....	286
7.3	Lunghezza e dimensione .....	295
7.4	Sottovalutare la dimensione di un insieme di dati .....	302
8.1	Tentare di modificare parametri di tipo primitivo.....	345
8.2	Mettere in ombra le variabili ( <i>shadowing</i> ) .....	364
8.3	Fare confusione con i punti.....	369
9.1	Dimenticarsi di dichiarare pubblici i metodi realizzati .....	387
9.2	Cercare di creare esemplari di un'interfaccia.....	390
9.3	Realizzare un metodo modificandone il tipo dei parametri .....	407
9.4	Dimenticare di associare un ricevitore .....	413
9.5	Per impostazione predefinita, i componenti hanno dimensioni nulle .....	414
9.6	Dimenticarsi di ridisegnare.....	417
10.1	Confondere superclassi e sottoclassi .....	435
10.2	Mettere in ombra variabili di esemplare.....	436
10.3	Sovrascrivere accidentalmente .....	440
10.4	Sbagliare nell'invocare metodi della superclasse .....	441
10.5	Sovrascrivere metodi rendendoli meno accessibili .....	450
10.6	Sovrascrivere il metodo <code>equals</code> con un tipo di parametro errato .....	467
11.1	Barre rovesciate ( <i>backslash</i> ) nei nomi di file .....	484
11.2	Costruire uno <code>Scanner</code> per leggere una stringa.....	484
12.1	Ricorsione infinita .....	527
12.2	Effettuare il tracciamento dell'esecuzione di metodi ricorsivi .....	528

13.1	Il metodo <code>compareTo</code> può restituire qualsiasi valore intero, non solo -1, 0 o 1 .....	589
15.1	Dimenticarsi di sovrascrivere <code>hashCode</code> .....	653
16.1	Genericità e ereditarietà .....	W-10
16.2	Usare tipi generici in un contesto statico .....	W-14

## Esempi completi



1.1	Sviluppo di un algoritmo per posare piastrelle su un pavimento .....	27
2.1	Quanti giorni sono trascorsi dalla vostra nascita? .....	55
2.2	Lavorare con immagini .....	57
3.1	Realizzare un semplice menu .....	102
4.1	Calcolare il volume e l'area della superficie totale di una piramide .....	151
4.2	Estrazione delle iniziali .....	164
5.1	Estrazione del carattere centrale .....	188
6.1	Elaborazione di carte di credito .....	251
6.2	Manipolare i pixel in un'immagine .....	258
6.3	Un esempio di sessione di debugging .....	271
7.1	Tirare i dadi .....	316
7.2	Una tabella con la popolazione mondiale .....	327
9.1	Analizzare sequenze di numeri .....	391
10.1	Progettare una gerarchia di dipendenti per l'elaborazione delle buste paga .....	457
11.1	Analizzare nomi di bambini .....	496
12.1	Cercare file .....	533
14.1	Una calcolatrice in notazione polacca inversa (RPN) .....	621
15.1	Determinare la frequenza di parole in un testo .....	640



## Note di cronaca

1.1	L'ENIAC e gli albori dell'informatica .....	7
2.1	I mainframe: quando i dinosauri dominavano la terra .....	65
2.2	L'evoluzione di Internet .....	78
3.1	Apparati per il voto elettronico .....	114
3.2	La grafica al calcolatore .....	122
4.1	Errore nel calcolo in virgola mobile nel Pentium .....	133
4.2	Alfabeti internazionali .....	163
5.1	Intelligenza artificiale .....	210
6.1	Il primo <i>bug</i> .....	276
7.1	Uno dei primi <i>worm</i> di Internet .....	289
7.2	Le vicende del Therac-25 .....	323
8.1	Lo sviluppo esplosivo dei personal computer .....	372
9.1	Sistemi operativi .....	402
9.2	Linguaggi di programmazione .....	422
10.1	Linguaggi di scripting .....	474
11.1	L'incidente del razzo Ariane .....	518
12.1	I limiti del calcolo automatico .....	550
13.1	Il primo programmatore .....	582

14.1	Standardizzazione.....	618
14.2	Notazione polacca inversa .....	621
15.1	Pirateria del software.....	685

## Sintassi di Java

1.1	Invocazione di metodo .....	13
2.1	Dichiarazione di variabile.....	38
2.2	Assegnazione.....	40
2.3	Costruzione di oggetti .....	47
2.4	Importazione di una classe da un pacchetto.....	52
3.1	Dichiarazione di variabile di esemplare.....	85
3.2	Dichiarazione di classe .....	97
3.3	Dichiarazione di metodo.....	97
4.1	Dichiarazione di costante .....	134
4.2	Cast .....	142
4.3	Invocazione di metodo statico.....	148
5.1	L'enunciato <b>if</b> .....	176
5.2	Confronti .....	181
5.3	Dichiarazione di tipo enumerativo .....	202
6.1	L'enunciato <b>while</b> .....	224
6.2	L'enunciato <b>for</b> .....	234
7.1	Array .....	285
7.2	Vettori.....	292
7.3	Il ciclo <b>for</b> esteso .....	299
8.1	Asserzione.....	351
8.2	Specifica di pacchetto .....	366
9.1	Dichiarazione di interfaccia.....	383
9.2	Implementazione di interfaccia .....	385
10.1	Ereditarietà .....	434
10.2	Invoke un metodo della superclasse.....	439
10.3	Invoke un costruttore della superclasse .....	442
10.4	L'operatore <b>instanceof</b> .....	446
11.1	Lanciare un'eccezione .....	502
11.2	La clausola <b>throws</b> .....	506
11.3	Catturare eccezioni .....	508
11.4	La clausola <b>finally</b> .....	510
16.1	Dichiarazione di una classe generica.....	W-5
16.2	Dichiarazione di un metodo generico .....	W-9

# Presentazione della quinta edizione italiana

La quinta edizione italiana del testo di Cay Horstmann si allinea con la sesta edizione del testo originale, *Java Concepts*, migliorata e rorganizzata dall'Autore che, tra le altre cose, ha aggiornato i contenuti alla recentissima versione 7 dell'ambiente Java, pur mantenendo la piena compatibilità di tutti gli esempi con le versioni 5 e 6, ancora molto utilizzate.

L'Autore ha realizzato una serie di “casi pratici” completi, che guidano il lettore neofita alla soluzione progettuale di un problema attraverso fasi successive e ben delineate, suddividendo sempre in modo appropriato le responsabilità. Per perseguire questo obiettivo, sono state introdotte nuove sezioni speciali di tipo “Esempi completi”, che senza dubbio agevoleranno lo studente nel suo percorso di apprendimento.

Non mancano interessanti novità nei primi capitoli, sempre volte a rendere graduale e progressivo l'apprendimento della progettazione a oggetti: uno dei segni distintivi della didattica proposta da Horstmann fin dalle prime edizioni. Tra le tante, mi preme evidenziare l'introduzione della descrizione di algoritmi mediante pseudocodice, colmando così quella che forse era una delle poche lacune delle versioni precedenti. Da notare anche il miglioramento apportato ad alcune delle sezioni dedicate agli array e ai relativi algoritmi, in modo da delineare una visione più organica dell'argomento, mentre i capitoli conclusivi, dedicati alle strutture dati, sono stati rivisti soltanto in modo marginale.

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 da tagliare.

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, tanto da spingere a una scelta editoriale innovativa, che ha evitato ulteriori oneri per la versione a stampa e propone l'accesso gratuito, sul sito web dedicato al libro, a un file PDF contenente, appunto, tutti gli esercizi di progettazione. Si tratta di una scelta a mio parere vincente, dal momento che tali esercizi vanno necessariamente svolti al calcolatore: potendoli analizzare sullo schermo del PC, ci si potrà esercitare anche senza avere il libro a portata di mano.

Rimangono, invece, nella versione a stampa gli "esercizi di ripasso" al termine di ogni capitolo: consentono allo studente di fare il punto sul livello della propria preparazione e sul raggiungimento degli obiettivi di apprendimento previsti, prima di passare al capitolo successivo e allo svolgimento dei più complessi e articolati problemi di programmazione.

*Marcello Dalpasso*

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

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

*Università degli Studi di Padova*

# Prefazione

Questo è un testo introduttivo di informatica che focalizza l'attenzione sui principi della programmazione e sull'ingegneria del software, seguendo alcune linee guida.

- **Insegnare gli oggetti 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.

- **Insistere su pratiche ingegneristiche adeguate.**

Concentrando l'attenzione sullo sviluppo orientato al collaudo, si incoraggiano gli studenti a collaudare i propri programmi in modo sistematico; analogamente, la presentazione degli errori più frequenti e di molti consigli utili per migliorare la qualità del software agevola lo sviluppo di buone abitudini di programmazione.

- **Assistere gli studenti con guide mirate e esempi completi.**

Spesso i programmatore principianti chiedono: "Come posso cominciare? 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 comuni, seguite dalla presentazione di ulteriori “Esempi completi”.

- **Concentrarsi sugli elementi 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, che danno soltanto l'illusione di aver compreso. In questo libro, gli aspetti essenziali di ciascun argomento sono presentati con trattazioni di 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.

- **Usare il linguaggio Java standard.**

Il libro insegna il linguaggio Java standard, non un ambiente speciale, dedicato all'apprendimento. Il livello di approfondimento, nella presentazione del linguaggio, della libreria e degli strumenti, è tale da consentire la soluzione di problemi di programmazione reali.

- **Presentare la programmazione grafica in un percorso opzionale.**

Le forme grafiche sono splendidi esempi di oggetti e molti studenti amano scrivere programmi grafici o, comunque, con interfaccia utente grafica. Se lo si desidera, questi argomenti possono essere integrati nella trattazione, utilizzando il materiale che si trova alla fine dei Capitoli 2, 3, 9 e 10.

## Le novità di questa edizione

Giunto alla sesta edizione (e quinta edizione italiana), il libro è stato nuovamente aggiornato e rivisto con cura, migliorando alcune caratteristiche e introducendone di nuove.

### Maggiore aiuto ai programmatori principianti

- I “Consigli pratici” sono stati aggiornati ed espansi, aggiungendone quattro. Quindi, nuovi “Esempi completi” guidano gli studenti passo dopo passo alla soluzione di problemi complessi e interessanti.
- È stata migliorata la trattazione della progettazione di algoritmi, con l'uso di pseudocodice fin dal Capitolo 1.
- Sono stati rivisti tutti i capitoli, in modo da focalizzare ciascun paragrafo su uno specifico obiettivo di apprendimento, secondo la filosofia dei *learning object*, che si riflette anche in una nuova organizzazione del riassunto, presente al termine di ogni capitolo per aiutare gli studenti a valutare i propri progressi.

### Discussione di esempi

- I diagrammi sintattici sono stati rivisitati graficamente, introducendo esempi tipici che attirino l'attenzione degli studenti verso gli elementi chiave della sintassi. Ulteriori annotazioni evidenziano casi speciali, errori comuni e buone pratiche relative a quella particolare sintassi.

- Sono state introdotte nuove tabelle di esempi per presentare in modo chiaro e compatto una varietà di casi tipici e speciali. Ogni esempio è accompagnato da una breve nota esplicativa sul suo utilizzo e sui valori che ne risultano.
- L'introduzione degli oggetti è stata resa ancor più graduale, con esempi aggiuntivi e approfondimenti fin dai primi capitoli.

## Aggiornamento a Java 7

- Le caratteristiche introdotte nella versione 7 di Java sono trattate in alcuni “Argomenti avanzati”, in modo che gli studenti le possano approfondire. In questa edizione, usiamo ancora Java 5 e 6 per la trattazione principale.

## Più opportunità per esercitarsi

- Il vasto insieme di esercizi, graduati per tipologia e difficoltà, è stato ulteriormente espanso e migliorato. Gli esercizi relativi alla grafica e al collaudo (*testing*) sono poi ulteriormente evidenziati mediante, rispettivamente, la lettera G o T. Al termine di ogni capitolo sono presenti esercizi di ripasso, mentre i più completi esercizi e progetti di programmazione si trovano nel sito web dedicato al libro.

## 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 era caratteristica dell'edizione precedente.

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

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

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.

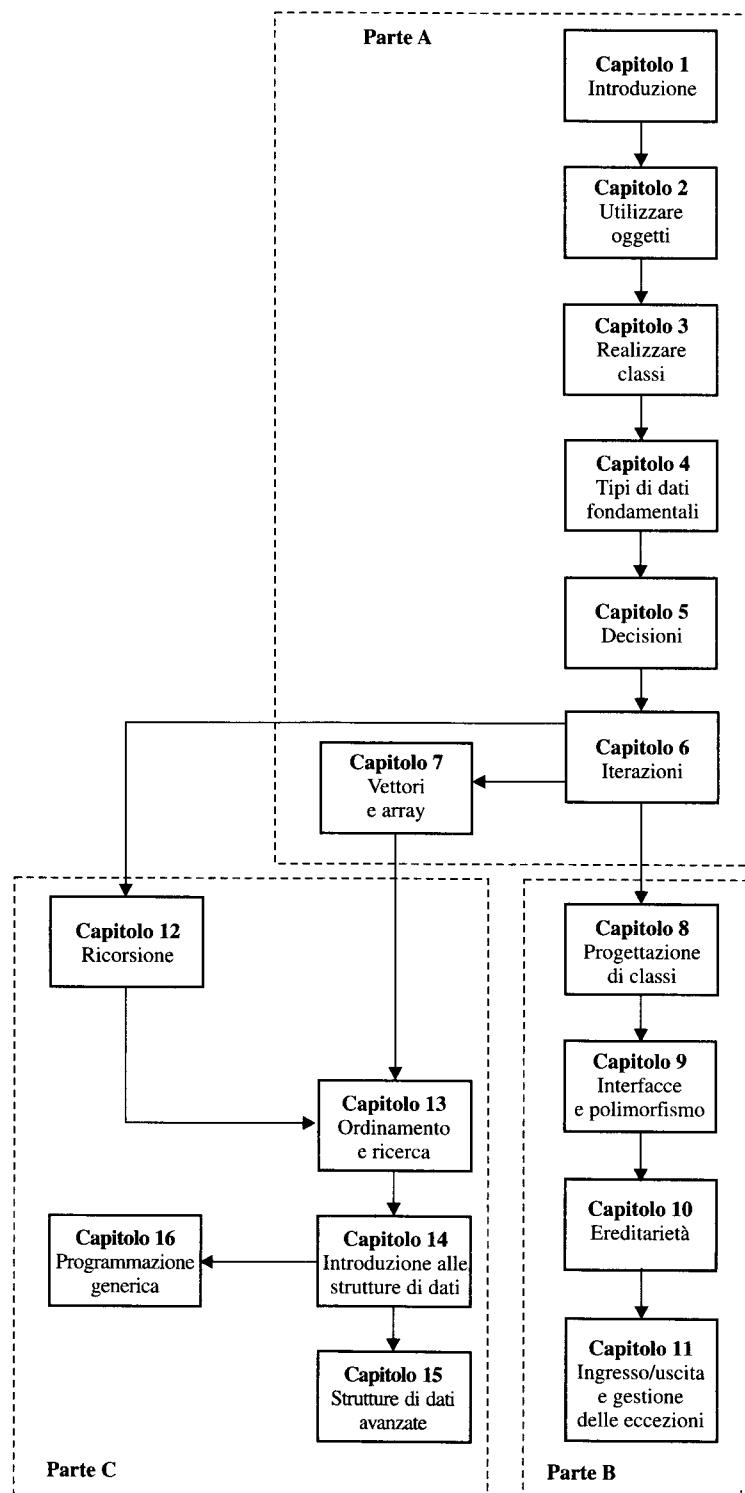
### Parte B: Progettazione orientata agli oggetti (Capitoli da 8 a 11)

Il Capitolo 8 si occupa della progettazione di classi in modo più sistematico e introduce un sottoinsieme semplificato della notazione UML.

La discussione sul polimorfismo e l'ereditarietà è suddivisa in due capitoli: il Capitolo 9 tratta le interfacce e il polimorfismo, mentre il Capitolo 10 si occupa dell'ereditarietà vera e propria. L'introduzione delle interfacce prima dell'ereditarietà è utile per un motivo fondamentale: gli studenti vedono immediatamente il polimorfismo prima di dover affrontare dettagli tecnici complessi, come la costruzione della superclasse.

La gestione delle eccezioni (la cui gerarchia costituisce un ottimo esempio di ereditarietà) e gli strumenti basilari per la gestione dei file di testo sono argomento del Capitolo 11.

**Figura 1**  
Dipendenze fra i capitoli



## Parte C: Algoritmi e strutture di dati (Capitoli da 12 a 16)

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, pile e code. Come si può vedere nella Figura 1, tali argomenti possono essere trattati, in alternativa, dopo il Capitolo 7.

La ricorsione viene presentata da un punto di vista orientato agli oggetti: un oggetto che risolve un problema ricorsivamente costruisce un altro oggetto della stessa classe che risolve un problema più semplice. Fare in modo che questo nuovo oggetto svolga il compito più semplice è molto più plausibile per gli studenti rispetto a una funzione che invoca se stessa.

Il Capitolo 13 si occupa degli algoritmi di ordinamento fondamentali e introduce l'analisi delle prestazioni mediante la notazione O-grande. Il Capitolo 14 tratta liste concatenate, pile e code, sia come tipi di dati astratti sia come realizzazioni concrete presenti nella libreria standard di Java. Il Capitolo 15 presenta alcune strutture di dati avanzate: tabelle hash, alberi di ricerca binari e heap.

Infine, il Capitolo 16, 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

L'Appendice A contiene le risposte alle domande di auto-valutazione presenti in tutti i capitoli, mentre le appendici seguenti contengono materiale di riferimento 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.

## 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, e possono essere utilizzate 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 nell'Appendice A.

Molte (nuove) tabelle agevolano il compito dei principianti, fornendo esempi concreti, per un veloce ripasso delle caratteristiche salienti 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.

Esistono, poi, sette diverse tipologie di note inserite nel testo, evidenziate in modo particolare per non interrompere il flusso del materiale principale. Alcune sono piuttosto brevi, altre occupano più di una pagina: ad ogni argomento è stato assegnato lo spazio necessario per una piena e convincente spiegazione, piuttosto che tentare di farle rientrare in “consigli” di un unico paragrafo.

- 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**, novità di questa edizione, applicano i Consigli pratici a un esempio diverso, illustrando come quei consigli siano effettivamente assai utili per pianificare, realizzare e collaudare 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 rimediavarvi.
- I **Consigli per la qualità** illustrano buone abitudini di programmazione. Dato che molti di essi richiedono uno sforzo iniziale per l’apprendimento, queste note danno attente motivazioni per i consigli stessi e spiegano come lo sforzo verrà ripagato in seguito.
- I **Consigli per la produttività** insegnano agli studenti come usare in modo più efficace gli strumenti e il tempo che hanno a disposizione, spingendoli a essere più produttivi.
- Gli **Argomenti avanzati** trattano materiale non essenziale o più complesso, oltre alle novità relative a Java 7.
- Le **Note di cronaca** forniscono informazioni storiche e sociali sull’informatica.

## Sito web dedicato al libro

All’indirizzo <http://www.apogeonline.com/libri/9788850329564/scheda> 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: esercizi e progetti di programmazione relativi a ciascun capitolo del libro, un capitolo dedicato alla programmazione generica e appendici che riportano un utile compendio sintattico del linguaggio Java e linee guida per la scrittura del codice sorgente.

## Ringraziamenti

I miei ringraziamenti vanno a Beth Golub, Lauren Sapira, Andre Legaspi, Don Fowley, Mike Berlin, Janet Foxman, Lisa Gee e Bud Peters di John Wiley & Sons, oltre a Vickie Piercy 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 Suzanne Dietrich, Rick Giles, Kathy Liszka, Stephanie Smullen, Julius Dichter, Patricia McDermott-Wells e David Woolbright per la 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:

Ian Barland, *Radford University*  
Rick Birney, *Arizona State University*  
Paul Bladek, *Edmonds Community College*  
Robert P. Burton, *Brigham Young University*  
Teresa Cole, *Boise State University*  
Geoffrey Decker, *Northern Illinois University*  
Eman El-Sheikh, *University of West Florida*  
David Freer, *Miami Dade College*  
Ahmad Ghafarian, *North Georgia College & State University*  
Norman Jacobson, *University of California, Irvine*  
Mugdha Khaladkar, *New Jersey Institute of Technology*  
Hong Lin, *University of Houston, Downtown*  
Jeanna Matthews, *Clarkson University*  
Sandeep R. Mitra, *State University of New York, Brockport*  
Parviz Partow-Navid, *California State University, Los Angeles*  
Jim Perry, *Ulster County Community College*  
Kai Qian, *Southern Polytechnic State University*  
Cyndi Rader, *Colorado School of Mines*  
Chaman Lal Sabharwal, *Missouri University of Science and Technology*  
John Santore, *Bridgewater State College*  
Stephanie Smullen, *University of Tennessee, Chattanooga*  
Monica Sweat, *Georgia Institute of Technology*  
Shannon Tauro, *University of California, Irvine*  
Russell Tessier, *University of Massachusetts, Amherst*  
Jonathan L. Tolstedt, *North Dakota State University*  
David Vineyard, *Kettering University*  
Lea Wittie, *Bucknell University*

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:

Tim Andersen, *Boise State University*  
Ivan Bajic, *San Diego State University*  
Ted Bangay, *Sheridan Institute of Technology*  
George Basham, *Franklin University*  
Sambit Bhattacharya, *Fayetteville State University*  
Joseph Bowbeer, *Vizrea Corporation*  
Timothy A. Budd, *Oregon State University*  
Frank Butt, *IBM*  
Jerry Cain, *Stanford University*  
Adam Cannon, *Columbia University*  
Nancy Chase, *Gonzaga University*  
Archana Chidanandan, *Rose-Hulman Institute of Technology*  
Vincent Cicirello, *The Richard Stockton College of New Jersey*  
Deborah Coleman, *Rochester Institute of Technology*  
Valentino Crespi, *California State University, Los Angeles*  
Jim Cross, *Auburn University*  
Russell Deaton, *University of Arkansas*  
H. E. Dunsomore, *Purdue University*  
Robert Duvall, *Duke University*  
Henry A. Etlinger, *Rochester Institute of Technology*  
John Fendrich, *Bradley University*  
John Fulton, *Franklin University*  
David Geary, *Sabreware, Inc.*  
Margaret Geroch, *Wheeling Jesuit University*  
Rick Giles, *Acadia University*  
Stacey Grasso, *College of San Mateo*  
Jianchao Han, *California State University, Dominguez Hills*  
Lisa Hansen, *Western New England College*  
Elliotte Harold  
Eileen Head, *Binghamton University*  
Cecily Heiner, *University of Utah*  
Brian Howard, *DePauw University*  
Lubomir Ivanov, *Iona College*  
Curt Jones, *Bloomsburg University*  
Aaron Keen, *California Polytechnic State University, San Luis Obispo*  
Elliot Koffman, *Temple University*

Kathy Liszka, *University of Akron*  
Hunter Lloyd, *Montana State University*  
Youmin Lu, *Bloomsburg University*  
John S. Mallozzi, *Iona College*  
John Martin, *North Dakota State University*  
Scott McElfresh, *Carnegie Mellon University*  
Joan McGrory, *Christian Brothers University*  
Carolyn Miller, *North Carolina State University*  
Teng Moh, *San Jose State University*  
John Moore, *The Citadel*  
Faye Navabi, *Arizona State University*  
Kevin O'Gorman, *California Polytechnic State University, San Luis Obispo*  
Michael Olan, *Richard Stockton College*  
Kevin Parker, *Idaho State University*  
Cornel Pokorny, *California Polytechnic State University, San Luis Obispo*  
Roger Priebe, *University of Texas, Austin*  
C. Robert Putnam, *California State University, Northridge*  
Neil Rankin, *Worcester Polytechnic Institute*  
Brad Rippe, *Fullerton College*  
Pedro I. Rivera Vega, *University of Puerto Rico, Mayaguez*  
Daniel Rogers, *SUNY Brockport*  
Carolyn Schauble, *Colorado State University*  
Christian Shin, *SUNY Geneseo*  
Jeffrey Six, *University of Delaware*  
Don Slater, *Carnegie Mellon University*  
Ken Slonneger, *University of Iowa*  
Peter Stanchev, *Kettering University*  
Ron Taylor, *Wright State University*  
Joseph Vybihal, *McGill University*  
Xiaoming Wei, *Iona College*  
Todd Whittaker, *Franklin University*  
Robert Willhoft, *Roberts Wesleyan College*  
David Womack, *University of Texas at San Antonio*  
Catherine Wyman, *DeVry University*  
Arthur Yanushka, *Christian Brothers University*  
Salih Yurttas, *Texas A & M University*

# 1

## Introduzione

### Obiettivi del capitolo

- Comprendere il significato dell'attività di programmazione
- Imparare a riconoscere le componenti più importanti dell'architettura dei computer
- Comprendere la distinzione fra linguaggi macchina e linguaggi di programmazione di alto livello
- Acquisire familiarità con la struttura di semplici programmi Java
- Compilare ed eseguire il primo programma Java
- Riconoscere gli errori che si manifestano durante la compilazione e durante l'esecuzione
- Scrivere algoritmi facendo uso di pseudocodice

Questo capitolo ha l'obiettivo di rendere familiare il concetto di programmazione, facendo una panoramica sull'architettura di un computer e illustrando le differenze tra codice macchina e linguaggi di programmazione di alto livello. Vedrete, poi, come compilare ed eseguire il vostro primo programma Java, imparando a diagnosticare gli errori che possono accadere quando si compila o si esegue un programma. Infine, imparerete a descrivere semplici algoritmi facendo uso di pseudocodice.

## 1.1 Che cos'è la programmazione?

Probabilmente avete già usato un computer, per lavoro o per svago: inoltre persone utilizzano i computer per attività quotidiane, come calcolare il saldo di un conto corrente bancario o scrivere un elaborato. I computer sono ottimi per questi lavori, perché possono gestire operazioni ripetitive, come sommare numeri o inserire parole in una pagina, senza annoiarsi o stancarsi. I computer sono anche ottimi strumenti per giocare, perché possono riprodurre sequenze di suoni e di immagini, coinvolgendo l'utente umano nel processo.

La flessibilità di un computer è un fenomeno davvero affascinante: la stessa macchina può calcolare il saldo di un conto corrente, stampare una relazione ed eseguire un gioco. Al contrario, altre macchine svolgono una gamma di attività più ristretta: un'automobile viaggia, un tostapane tosta il pane.

Per ottenere tale flessibilità, il computer deve essere *programmato* per svolgere ciascuna attività. 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. I programmi sono sequenze di istruzioni e di decisioni che il computer esegue per svolgere un'attività. Un programma calcola il saldo del conto corrente; un altro programma, probabilmente progettato e realizzato da una società diversa, elabora testi; infine, un terzo programma consente di giocare.

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

- accendi un punto rosso in questa posizione dello schermo;
- estrai un numero da questa posizione della memoria;
- somma questi due numeri;
- se questo valore è negativo, prosegui da quella istruzione.

Un programma descrive al computer, nei minimi dettagli, la sequenza di passaggi necessari per portare a termine un determinato compito ed è composto da un numero enorme di operazioni elementari, che vengono eseguite dal computer a una velocità elevatissima. Il computer non è dotato di intelligenza: esegue semplicemente sequenze di istruzioni che sono state precedentemente predisposte.

Per usare un computer non è necessaria alcuna conoscenza di programmazione. Quando scrivete una relazione mediante un *word processor*, cioè un elaboratore di testi, utilizzate un programma che qualcun altro ha già sviluppato e che, quindi, è pronto per l'uso. Questo è proprio ciò che ci si aspetta: infatti, potete guidare un'automobile senza essere un meccanico e tostare il pane senza essere un elettricista.

Uno degli obiettivi principali di questo libro è quello di insegnarvi a progettare e realizzare programmi per computer: imparerete a enunciare istruzioni per tutti i compiti che i vostri programmi devono eseguire.

Tenete bene a mente che la programmazione di un sofisticato gioco per computer o di un elaboratore di testi richiede una squadra di molti programmatore altamente specializzati, di artisti grafici e di persone con altre elevate competenze. I vostri primi esercizi di programmazione saranno più elementari, ma i concetti e le competenze che

**Un computer, per svolgere compiti specifici, deve essere programmato: compiti diversi richiedono programmi diversi.**

**Un programma per computer esegue una sequenza di istruzioni elementari a velocità molto elevata.**

**Un programma contiene le sequenze di istruzioni necessarie a raggiungere tutti i suoi obiettivi..**

apprenderete in questo corso costituiscono comunque una base importante, anche se non potrete pensare di poter produrre immediatamente software professionale. Solitamente, un percorso universitario completo in informatica o in ingegneria del software viene completato in tre o quattro anni e questo testo è stato pensato per essere utilizzato in uno dei primi insegnamenti di tale percorso.

Molti studenti scoprono che anche semplici attività di programmazione sono molto stimolanti: è un'esperienza assai gratificante vedere il computer svolgere con precisione e rapidità un'attività che avrebbe richiesto ore di fatica.



## Auto-valutazione

1. Cosa si utilizza per riprodurre un CD musicale con un computer?
2. Perché un riproduttore di CD è meno flessibile di un computer?
3. Un programma per calcolatore è in grado, di propria iniziativa, di eseguire i suoi compiti in un modo migliore di quello identificato dai suoi programmati?

## 1.2 L'anatomia di un computer

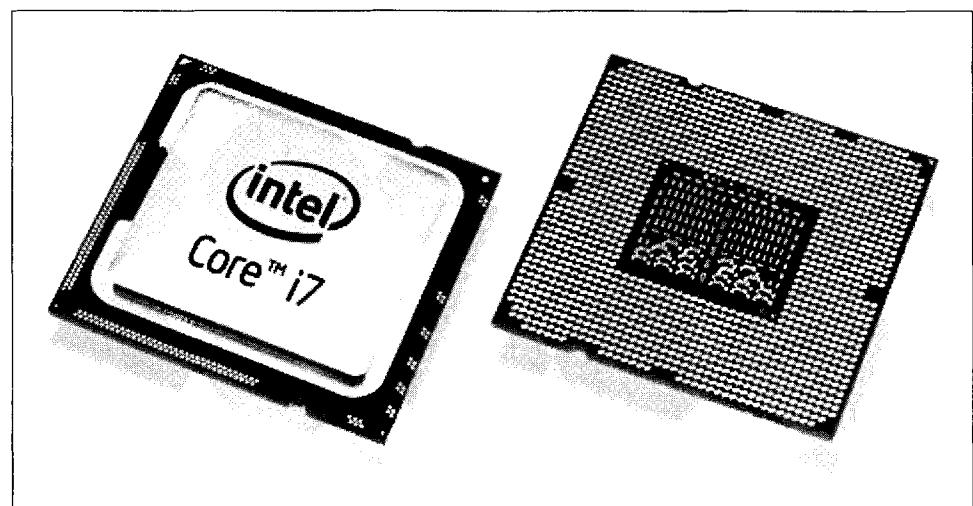
Per comprendere il processo di programmazione, è necessario conoscere almeno per grandi linee gli elementi costitutivi di un computer. Pertanto, daremo uno sguardo a un personal computer: le macchine di maggiori dimensioni hanno componenti più veloci, più grandi o più potenti, ma hanno sostanzialmente la stessa struttura.

Nel cuore del computer si trova l'*unità centrale di elaborazione* (CPU, *central processing unit*, in Figura 1), che è formata da un unico *chip* o da un piccolo numero di chip. Un chip (o circuito integrato) è un componente con connettori metallici e collegamenti interni, costituito principalmente di silicio e alloggiato in un contenitore di plastica o di metallo. Nel chip di una CPU, i collegamenti interni sono estremamente complicati: ad esempio, il processore Intel Core (una CPU per personal computer poco costosa e molto diffusa) contiene parecchie centinaia di milioni di elementi strutturali, detti *transistor*, gli elementi

L'unità centrale di elaborazione (CPU, central processing unit) costituisce il cuore del computer.

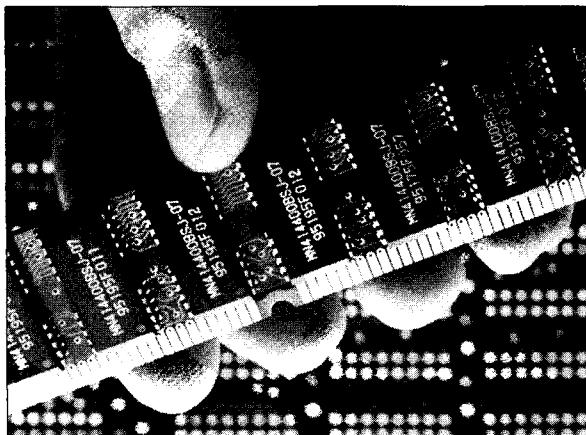
**Figura 1**

Unità centrale di elaborazione (CPU)



**Figura 2**

Un modulo con chip di memoria



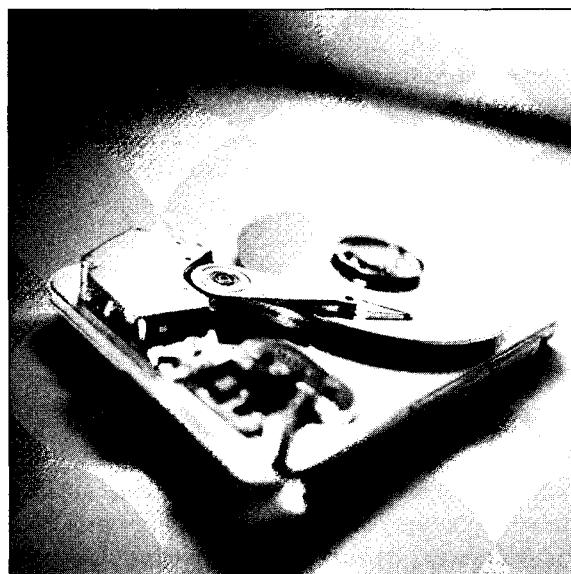
I dati e i programmi sono memorizzati nella memoria principale (semplicemente, memoria) e nella memoria secondaria (ad esempio, un disco rigido)

che consentono a un segnale elettrico di controllare altri segnali elettrici, rendendo così possibile il calcolo automatico. La CPU individua ed esegue le istruzioni del programma; effettua le operazioni aritmetiche, quali addizioni, sottrazioni, moltiplicazioni e divisioni; reperisce dati dalla memoria esterna o dalle apparecchiature periferiche, oppure ve li memorizza.

Il computer immagazzina dati e programmi nella memoria, di cui esistono due tipologie. La *memoria principale*, detta anche *memoria ad accesso casuale* (RAM, *random access memory*) o semplicemente *memoria*, è veloce, ma costosa; è costituita da chip di memoria (Figura 2). La memoria principale perde tutti i dati quando il computer viene spento. La *memoria secondaria*, che generalmente è un *disco rigido* (*hard disk*, in Figura 3), consente una registrazione dei dati meno costosa e che perdura anche in assenza di elettricità. Un disco rigido è formato da piatti rotanti, rivestiti da materiale magnetico, e da testine di lettura/scrittura, in grado di leggere e di modificare il flusso magnetico sui piatti.

**Figura 3**

Un disco rigido



Ancuni computer sono unità autosufficienti, mentre altri sono connessi tra loro tramite *reti*. I computer di casa di solito vengono connessi alla rete Internet saltuariamente, tramite una linea telefonica o una connessione a larga banda, mentre i computer di un laboratorio informatico sono probabilmente connessi a una rete locale in modo permanente. Attraverso le connessioni della rete, il computer può leggere programmi da una postazione centralizzata oppure inviare dati ad altri computer. Per l'utente di un computer connesso in rete, non sempre può essere semplice distinguere i dati che risiedono sulla propria macchina da quelli che vengono trasmessi attraverso la rete stessa.

La maggior parte dei computer ha dispositivi di memorizzazione *rimovibili*, che consentono l'accesso a dati o programmi che si trovano su supporti fisici come schede di memoria (o "chiavette") o dischi ottici.

Per interagire con un utente umano, un computer ha bisogno di altri dispositivi periferici: trasmette le informazioni mediante uno schermo di visualizzazione, altoparlanti e stampanti, mentre l'utente può inserire informazioni e impartire ordini al computer tramite una tastiera o un dispositivo di puntamento, quale un mouse.

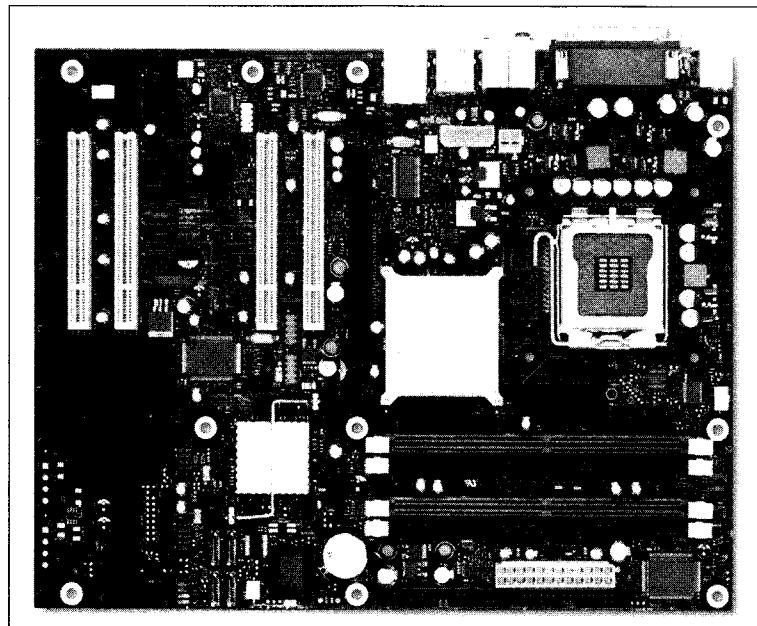
La CPU, la memoria RAM, l'elettronica che controlla il disco rigido e gli altri dispositivi sono interconnessi mediante un insieme di linee elettriche, che formano un *bus*. I dati transitano lungo il bus, dalla memoria del sistema e dai dispositivi periferici verso la CPU e viceversa. La Figura 4 mostra una *scheda principale* o "scheda madre" (*motherboard*), che contiene la CPU, la RAM e i connettori per i dispositivi periferici.

La Figura 5 presenta uno schema dell'architettura di un computer. Le istruzioni dei programmi e i relativi dati (quali testi, numeri, sequenze audio o video) sono conservati nel disco rigido, in un disco ottico (come un DVD) o in qualche punto di una rete. Quando si avvia un programma, lo si copia nella memoria, dove la CPU può leggerlo, un'istruzione per volta. A seconda delle direttive espresse da tali istruzioni, la CPU legge dati, li modifica e li scrive nuovamente nella memoria RAM o nella memoria secondaria;

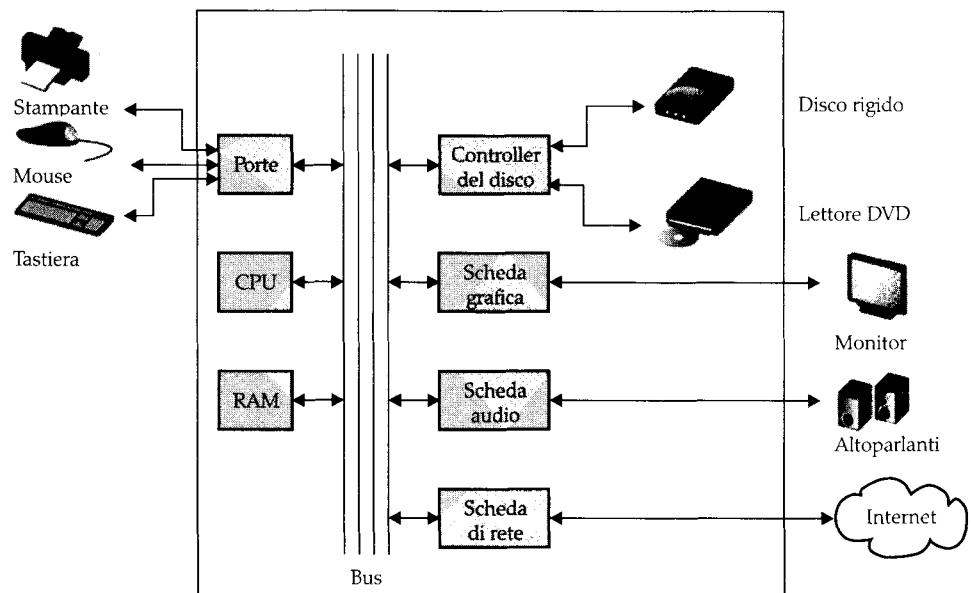
La CPU legge dalla memoria  
e istruzioni, che la fanno comunicare  
con la memoria principale,  
la memoria secondaria  
e i dispositivi periferici.

**Figura 4**

La scheda principale  
(*motherboard*)



**Figura 5**  
Diagramma schematico  
di un personal computer



alcune istruzioni del programma indurranno la CPU a interagire con lo schermo o con l'altoparlante. Poiché queste azioni si ripetono molte volte e a grande velocità, l'utente umano percepirà immagini e suoni. Analogamente, la CPU può inviare istruzioni a una stampante per imprimere sulla carta insiemi di punti molto vicini tra loro, che un utente umano riconoscerà come caratteri di un testo e figure. Alcune istruzioni ricevono i comandi dell'utente dalla tastiera o dal mouse: il programma esamina la natura di questi comandi ed esegue conseguentemente le istruzioni appropriate.



### Auto-valutazione

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

## 1.3 Passare da programmi leggibili al codice macchina

In generale, il codice macchina è specifico di ciascuna CPU, ma l'insieme delle istruzioni della JVM può essere eseguito da CPU diverse.

Al livello più basso, le istruzioni di un computer sono estremamente elementari. Il processore esegue *istruzioni macchina* e le CPU di produttori diversi, quali il processore Pentium di Intel o lo SPARC di Sun, hanno insiemi di istruzioni macchina differenti. Per fare in modo che le applicazioni Java siano eseguite senza modifiche da CPU diverse, i programmi Java contengono istruzioni macchina per una cosiddetta “macchina virtuale Java” (JVM, Java Virtual Machine), ovvero una CPU ideale che viene simulata da un programma in esecuzione sulla CPU effettiva.

Le istruzioni per le macchine reali e per quella virtuale sono molto semplici e possono essere eseguite assai rapidamente. Ecco una tipica sequenza di istruzioni macchina:

- Carica (cioè “leggi e copia all’interno della CPU”) il contenuto della posizione di memoria 40.
- Carica il valore 100.
- Se il primo valore è maggiore del secondo, prosegui eseguendo l’istruzione contenuta nella posizione di memoria 240.

In pratica, le istruzioni macchina conservate in memoria sono codificate sotto forma di numeri. Nella macchina virtuale Java, le istruzioni precedenti sono codificate mediante questa sequenza di numeri:

```
21 40
16 100
163 240
```

## IL CAFFÈ

### Note di cronaca 1.1

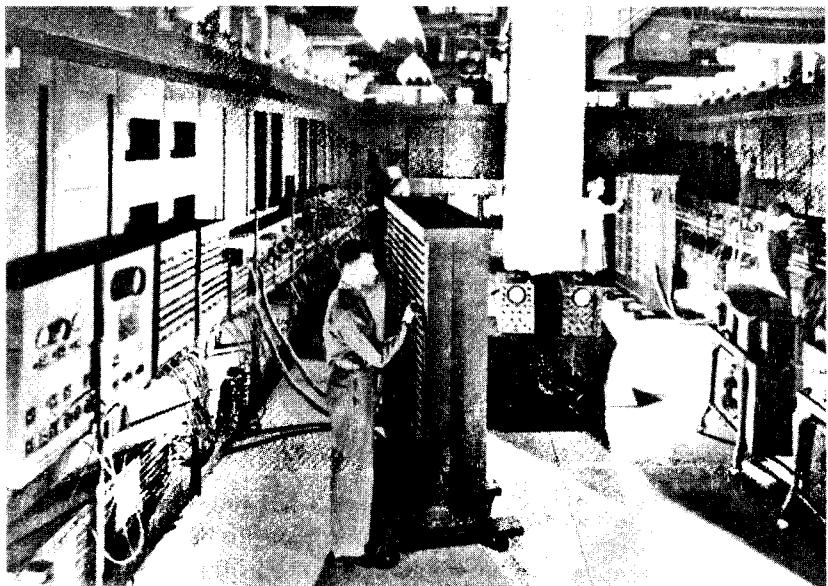
#### **ENIAC e gli albori dell’informatica**

L’ENIAC (Electronic Numerical Integrator And Computer, integratore numerico e calcolatore elettronico), fu il primo computer elettronico ad essere utilizzato. Fu progettato presso la University of Pennsylvania da J. Presper Eckert e John Mauchly e venne completato nel 1946, due anni prima dell’invenzione dei transistor. Il computer era ospitato in più armadi e usava circa 18 000 valvole termoioniche (tubi a vuoto), che si bruciavano al ritmo di molte al giorno e un inserviente, con un carrello della spesa pieno di valvole, girava continuamente per sostituire quelle difettose. Il computer veniva programmato collegando cavi su pannelli e ciascuna configurazione delle connessioni impostava il computer per un problema specifico. Per consentire al computer di lavorare su un problema diverso, occorreva cambiare la connessione dei cavi.

Il lavoro sull’ENIAC era finanziato dalla U.S. Navy, la Marina Militare degli Stati Uniti d’America, che

era interessata al calcolo di tavole balistiche che dovevano fornire la traiettoria di un proiettile in funzione della resistenza del vento, della velocità iniziale e delle condizioni atmosferiche. Per calcolare tali traiettorie, occorreva trovare le soluzioni numeriche di equazioni differenziali, da cui sortì il nome di “integratore numeri-

co”. Prima che venissero sviluppate macchine come l’ENIAC, questo tipo di lavoro veniva svolto da esseri umani e, fino agli anni ’50, il termine “computer” indicava proprio queste persone. Più tardi, l’ENIAC venne impiegato per scopi pacifici, come l’elaborazione dei dati del servizio demografico statunitense.



Quando la macchina virtuale legge questa sequenza di numeri, li decodifica ed esegue la serie di comandi corrispondenti.

Poiché le istruzioni macchina sono codificate come numeri, è difficile scrivere programmi in codice macchina.

I linguaggi ad alto livello permettono la descrizioni dei compiti da svolgere a un livello concettuale più elevato rispetto al codice macchina.

In che modo possiamo comunicare la sequenza di comandi al computer? Il metodo più diretto consiste nell'inserire i numeri effettivi nella sua memoria: di fatto, questo è il modo in cui lavoravano i primissimi computer. Tuttavia, un programma lungo è composto da migliaia di comandi singoli ed è noioso, nonché causa di errori, cercare i codici numerici di tutti i comandi per poi inserirli manualmente nella memoria. Come già detto precedentemente, i computer si prestano molto bene all'automazione delle attività noiose e inclini agli errori, e ai programmatore non occorre molto tempo per comprendere che potevano sfruttare i computer stessi per agevolare l'attività di programmazione.

A metà degli anni '50 iniziarono ad apparire linguaggi di programmazione ad alto livello: in questi linguaggi, il programmatore esprime in astratto l'operazione da compiere, mentre un programma speciale, detto *compilatore*, traduce la descrizione di alto livello nelle istruzioni macchina idonee per un processore specifico.

Per esempio, in Java, il linguaggio di programmazione ad alto livello che useremo in questo corso, potrete imparire l'istruzione seguente:

```
if (intRate > 100)
    System.out.println("Errore nel tasso di interesse");
```

L'istruzione significa "se il tasso d'interesse è maggiore di 100, visualizza un messaggio di errore". Poi, sarà compito del programma compilatore individuare la sequenza di caratteri `if (intRate > 100)` e tradurla in:

21 40 16 100 163 240 ...

Un compilatore traduce in codice macchina programmi scritti in un linguaggio di alto livello.

I compilatori sono programmi piuttosto sofisticati. Hanno il compito di tradurre enunciati logici, come l'enunciato `if`, in sequenze di calcoli, verifiche e salti all'interno del programma, oltre ad assegnare posizioni di memoria alle *variabili*, elementi di informazione identificati da nomi simbolici, come `intRate`. In questo corso, generalmente daremo per scontata la presenza di un compilatore. Se diventerete informatici professionisti, potrete sempre approfondire le tecniche di progettazione dei compilatori proseguendo nei vostri studi.



## Auto-valutazione

6. Qual è il codice corrispondente all'istruzione "Carica il contenuto della posizione di memoria 100" per la macchina virtuale Java?
7. Una persona che utilizza il computer per il normale lavoro d'ufficio userà mai un compilatore?

## 1.4 Il linguaggio di programmazione Java

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

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", per l'utilizzo in elettrodomestici, come gli apparecchi "intelligenti" per televisori (*set-top box*). Il linguaggio venne progettato per essere semplice e neutrale rispetto all'architettura, in modo da operare su hardware diversi. ma non si trovò mai alcun cliente per questa tecnologia.

Gosling racconta che nel 1994 la squadra si rese conto che: "Avremmo 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 che avevamo ottenuto: neutralità rispetto all'architettura, esecuzione in tempo reale, affidabilità, sicurezza". Java fu presentato a una folla entusiasta durante l'evento SunWorld del 1995.

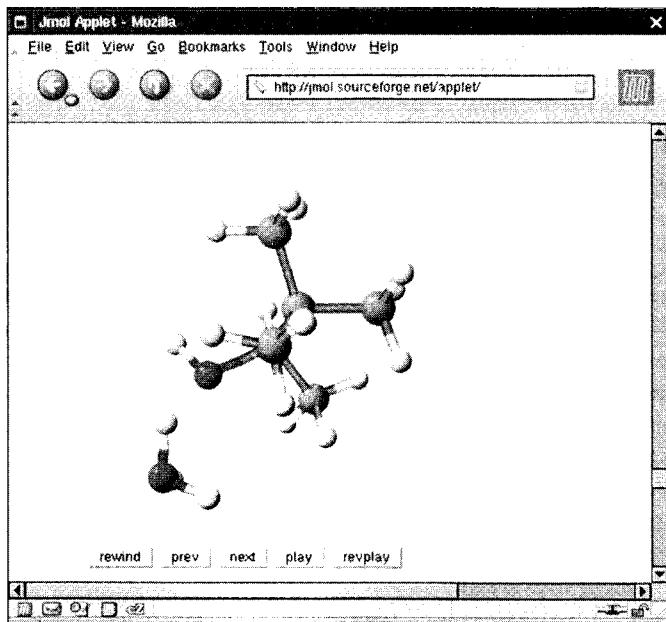
**Java fu progettato per essere sicuro e portatile, a vantaggio sia degli utenti di Internet sia degli studenti.**

Da quel giorno, il linguaggio Java si è imposto a un ritmo fenomenale. I programmati l'hanno adottato perché è più semplice del suo rivale che più gli assomiglia, 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 trasferibili ("portabili") da un sistema operativo all'altro, una funzionalità attesa ansiosamente da quanti volevano essere indipendenti da quei sistemi proprietari e decisamente osteggiata dai loro fornitori. Le due versioni della libreria, una "micro edition" e una "enterprise edition", rendono possibile il lavoro dei programmati in situazioni che vanno dai più piccoli dispositivi integrati, come le *smart card* e i telefoni cellulari, ai più grandi server di Internet.

Poiché Java è stato progettato per Internet, ha due qualità che lo rendono molto adatto per i principianti: sicurezza e trasferibilità. Se si visita una pagina Web che contiene codice Java (i cosiddetti *applet*, di cui vedete un esempio in Figura 6), questo viene eseguito automaticamente. È quindi assai importante poter contare sul fatto che gli applet siano intrinsecamente sicuri. Se un applet facesse qualcosa di male, come danneggiare dati o leggere informazioni personali sul vostro computer, vi trovereste in reale pericolo a ogni navigazione sul Web: un progettista senza scrupoli potrebbe impostare una pagina contenente codice pericoloso, che si attiverebbe sulla vostra macchina appena visitate la pagina. Al contrario, il linguaggio Java è dotato di molte caratteristiche di sicurezza per garantire che non si possano eseguire applet nocivi. Quale vantaggio ulteriore, queste caratteristiche aiutano anche a imparare il linguaggio più velocemente: la macchina vir-

**Figura 6**

- applet per visualizzare molecole, eseguito
- all'interno di un browser Web (<http://jmol.sourceforge.net/applet/>)



tuale di Java può cogliere molti tipi di errori da principiante e segnalarli accuratamente (per contro, nel linguaggio C++ molti errori da principiante producono semplicemente programmi che agiscono in modo arbitrario e disorientante).

L'altro vantaggio di Java è la trasferibilità o portabilità: lo stesso programma Java opererà senza bisogno di modifiche in ambiente Windows, UNIX, Linux o Macintosh. Anche questo è un requisito degli applet: quando si visita una pagina Web, il server Web che distribuisce il contenuto della pagina non ha idea di quale computer stiate utilizzando per visualizzarla, ma vi restituisce semplicemente il codice portabile che è stato generato dal compilatore Java e la macchina virtuale presente nel vostro computer esegue tale codice. Anche qui esiste un vantaggio per lo studente: non è necessario imparare a scrivere programmi per sistemi operativi diversi.

Allo stato attuale, Java si è imposto 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 curato affinché fosse veramente semplice da utilizzare per scrivere programmi 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 svantaggio per lo studente inesperto. Mentre imparate come programmare in Java, in alcuni casi vi dovrete accontentare di spiegazioni preliminari e dovrete attendere un capitolo successivo per avere tutti i particolari.

Java ha subito varie revisioni ed estensioni, come potete vedere nella Tabella 1. Si assume, qui, che abbiate a disposizione Java nella versione 5 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> esteso, auto-incapsulamento, enumerazioni
6	2006	Miglioramenti della libreria
7	2010	Piccole modifiche al linguaggio e miglioramenti della libreria

**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, in pochi mesi non potete sperare di imparare tutto di Java. Di per sé, il linguaggio Java è relativamente semplice, ma Java contiene un'ampia raccolta di *pacchetti di libreria*, che sono necessari per scrivere programmi utili. Si tratta di pacchetti per la grafica, per la costruzione di interfacce utente, 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 programmati esperti di Java non conoscono il contenuto di tutti i pacchetti, ma si limitano a usare quelli che servono per un particolare progetto.

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



## Auto-valutazione

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

## 1.5 La struttura di 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.

### File ch01/hello/HelloPrinter.java

```
public class HelloPrinter
{
    public static void main(String[] args)
    {
        // visualizza un messaggio di saluto sulla finestra di console
        System.out.println("Hello, World!");
    }
}
```

### Esecuzione del programma

Hello, World!

Nel paragrafo successivo vedrete come compilare ed eseguire questo programma, ma prima cerchiamo di capire come sia strutturato. La prima riga:

```
public class HelloPrinter
```

inizia una nuova *classe*. La classe è un concetto fondamentale in Java, che inizierete a studiare nel Capitolo 2. Ogni programma Java è costituito da una o più classi.

La parola *public* indica che la classe è utilizzabile "dal pubblico", cioè da qualunque punto interno al vostro programma. 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)
{
    ...
}
```

definisce un *metodo*, chiamato *main* ("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 2 vedrete come scriverli.

**Le classi sono i blocchi fondamentali**

**La costruzione di programmi**

**Java.**

**Per ogni applicazione Java esiste**

**una classe che ha il metodo main.**

**Quando l'applicazione viene**

**eseguita, vengono eseguite**

**e istruzioni del metodo main.**

La parola **static** e la dichiarazione **String[] args** verranno spiegate nei Capitoli 7 e 10. A questo punto del vostro apprendimento, dovreste considerare semplicemente

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

come una “infrastruttura” necessaria per scrivere qualsiasi programma Java.

La prima riga all’interno del metodo **main** è un *commento*:

```
// visualizza un messaggio di saluto sulla finestra di console
```

**Usate i commenti per aiutare i lettori umani a capire il vostro programma.**

Questo commento è stato inserito soltanto a beneficio di un lettore umano, per spiegare con maggior dettaglio l’effetto dell’enunciato seguente. Qualsiasi testo compreso tra **//** e la fine della riga viene totalmente ignorato dal compilatore. I commenti vengono usati per rendere il programma più chiaro ad altri programmatore o a se stessi.

Gli *enunciati* (o istruzioni) presenti nel *corpo* del metodo **main**, ovvero gli enunciati racchiusi fra le parentesi graffe **{ }**, sono eseguiti uno alla volta, in ordine. Ciascun enunciato termina con un carattere “punto e virgola”. Il nostro metodo ha un solo enunciato:

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

Questo enunciato visualizza (o “stampa”) una riga di testo, vale a dire “Hello, World!”. Tuttavia, esistono molti posti dove un programma può inserire questo testo: una finestra, un file, oppure un computer connesso alla rete dall’altra parte del mondo. Pertanto, occorre specificare che la destinazione di questa stringa è l’*uscita standard*, ovvero una finestra di terminale presente sullo schermo. In Java, la finestra di terminale in cui viene eseguito un programma è rappresentata da un oggetto chiamato **System.out**. Un *oggetto* è un’entità manipolabile all’interno di un programma.

In Java, ogni oggetto appartiene a una classe, la quale dichiara i metodi che specificano ciò che si può fare con l’oggetto stesso. L’oggetto **System.out** appartiene alla classe **PrintStream**, la quale ha un metodo, denominato **println**, per stampare una riga di testo.

Non c’è bisogno che realizziamo questo metodo, perché i programmatore che hanno scritto la libreria di Java l’hanno già fatto per noi: dobbiamo solamente invocarlo (o “chiamarlo”).

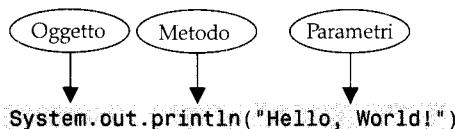
Quando invocate un metodo in Java, dovete specificare tre elementi, come potete vedere nella Figura 7:

1. L’oggetto che volete usare (in questo caso, **System.out**).
2. Il nome del metodo che volete invocare (in questo caso, **println**).
3. Una coppia di parentesi tonde, che racchiudono eventuali altre informazioni necessarie per il metodo (in questo caso, “Hello, World!”). Dal punto di vista tecnico, questa informazione viene chiamata *parametro* del metodo.

**Per invocare un metodo si specifica un oggetto, il nome del metodo e i suoi parametri.**

**Figura 7**

Invocazione di un metodo



Una stringa è una sequenza  
di caratteri racchiusa tra virgolette.

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

"Hello, World!"

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 stampare 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 le interpreti come istruzioni del programma.

Si possono stampare anche valori numerici. Per esempio, l'enunciato seguente visualizza il numero 7:

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

Il metodo `println` stampa 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!");`

stampa due righe di testo, in questo modo:

Hello  
World!

Esiste un secondo metodo, chiamato `print`, che potete utilizzare per stampare un elemento senza iniziare una nuova riga subito dopo.

## Sintassi di Java

### 1.1 Invocazione di metodo

#### Sintassi

`oggetto.nomeMetodo(parametri)`

#### Esempio

Il metodo viene invocato su questo oggetto.

Questo è il nome del metodo.

Questo parametro viene passato al metodo.

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

I parametri sono racchiusi tra parentesi tonde.  
Più parametri sono separati da virgole.

Per esempio, questi due enunciati:

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

visualizzano la singola riga seguente:

007



## Auto-valutazione

10. Come modifichereste il programma `HelloPrinter` in modo che visualizzi le parole "Hello," e "World!" in due righe consecutive?
11. Il programma continua a funzionare anche se la riga che inizia con `//` viene eliminata?
12. Cosa viene visualizzato dal seguente insieme di enunciati?

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



## Errori comuni 1.1

### Dimenticare i punti e virgola

In Java, ciascun enunciato deve terminare con un punto e virgola. Dimenticare di digitarlo è un errore frequente che disorienta il compilatore, dal momento che si basa sul punto e virgola per stabilire dove termina un enunciato e ne inizia un altro: il compilatore, infatti, non usa le interruzioni di riga o le parentesi graffe 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: scorrete tutti gli enunciati per verificare che terminino con un punto e virgola, proprio come controllereste che ciascuna frase in italiano termini con il punto.



## Argomenti avanzati 1.1

### Sintassi alternativa per i commenti

Java consente di scrivere i commenti in due modi diversi. Avete già appreso che il compilatore ignora qualsiasi cosa digitata fra `//` e la fine della riga. Allo stesso modo, il compilatore ignora qualsiasi testo racchiuso fra `/*` e `*/`.

```
/* Un semplice programma Java */
```

La doppia barra // di un commento è più facile da digitare se il testo trova posto in una sola riga, ma, se avete un commento che si estende oltre una riga o due, la forma /\* ... \*/ è più adatta:

```
/*
    Questo è un semplice programma Java che potete usare per provare
    il compilatore e la macchina virtuale.
*/
```

Sarebbe alquanto noioso inserire la doppia barra // all'inizio di ciascuna riga, per poi spostarla se il testo del commento cambia.

In questo libro useremo la doppia barra // per i commenti che si estendono su una sola riga e la forma /\* ... \*/ per quelli più lunghi. Se preferite, potete utilizzare sempre lo stile //. I lettori del vostro codice apprezzeranno *qualsiasi* commento, indipendentemente dallo stile utilizzato.

## 1.6 Compilare ed eseguire un programma Java

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  
ad esempio, un programma Java).

Java distingue tra maiuscolo  
e minuscolo.

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

Alcuni ambienti di sviluppo per Java sono molto facili da utilizzare. Basta inserire il codice in una finestra, premere un pulsante con il mouse per compilare e premerne un altro per eseguire il programma. I messaggi d'errore compaiono in una seconda finestra e il programma viene eseguito in una terza. Con un simile ambiente i dettagli del processo di compilazione rimangono nascosti, mentre in altri sistemi dovete eseguire tutti i passaggi manualmente, scrivendo in una finestra di comandi (*console* o *shell*).

Indipendentemente dall'ambiente di sviluppo che utiliziate, inizierete la vostra attività di programmazione scrivendo gli enunciati del programma. Il programma che si utilizza per scrivere e modificare il testo del programma si chiama *editor*. Far partire l'*editor* è il primo passo per la creazione di *qualsiasi* programma Java, come il programma Hello-Printer visto nel paragrafo precedente: create un nuovo file e chiamatelo *HelloPrinter.java* (se il vostro ambiente di programmazione vi chiede il nome del “progetto”, oltre al nome del file, chiamatelo *hello*). Copiate con precisione le istruzioni del programma, così come le trovate scritte nel libro.

Java *distingue fra maiuscolo e minuscolo*, quindi dovete inserire le lettere maiuscole e minuscole esattamente come appaiono nel testo: non si può digitare **MAIN** oppure **PrintLn**. Se non si sta attenti, si incorre in problemi (si vedano gli Errori comuni 1.2). Per contro, Java consente una *disposizione del testo a formato libero*. Per separare due parole si può usare un numero *qualsiasi* di spazi e/o di interruzioni di riga e in ciascuna riga potete inserire tutte le parole che volete:

```
public class HelloPrinter{public static void main(String[]
args){// visualizza un messaggio di saluto sulla finestra di console
System.out.println("Hello, World!");}}
```

Impaginate il vostro programma  
in modo da agevolarne la lettura.

Ovviamente, però, questa non è una buona idea: bisogna impaginare i programmi in modo chiaro, per agevolarne la lettura da parte di altri e di noi stessi. In questo libro

troverete molti consigli per una buona impaginazione dei programmi e l'Appendice G li riassume.

Passiamo ora all'esecuzione del programma: da qualche parte sullo schermo (Figure 8 e 9) comparirà il messaggio

Hello, World!

La posizione esatta del messaggio dipende dal vostro ambiente di programmazione.

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

L'esecuzione del programma avviene in due fasi (anche se alcuni ambienti di sviluppo li eseguono automaticamente entrambi quando si chiede l'esecuzione di un programma).

Il primo passo è la *compilazione* del programma. Il compilatore traduce il *codice sorgente* Java (gli enunciati che avete scritto) in *file di classi*, che contengono istruzioni per la macchina virtuale e altre informazioni necessarie per l'esecuzione. I file di classi hanno

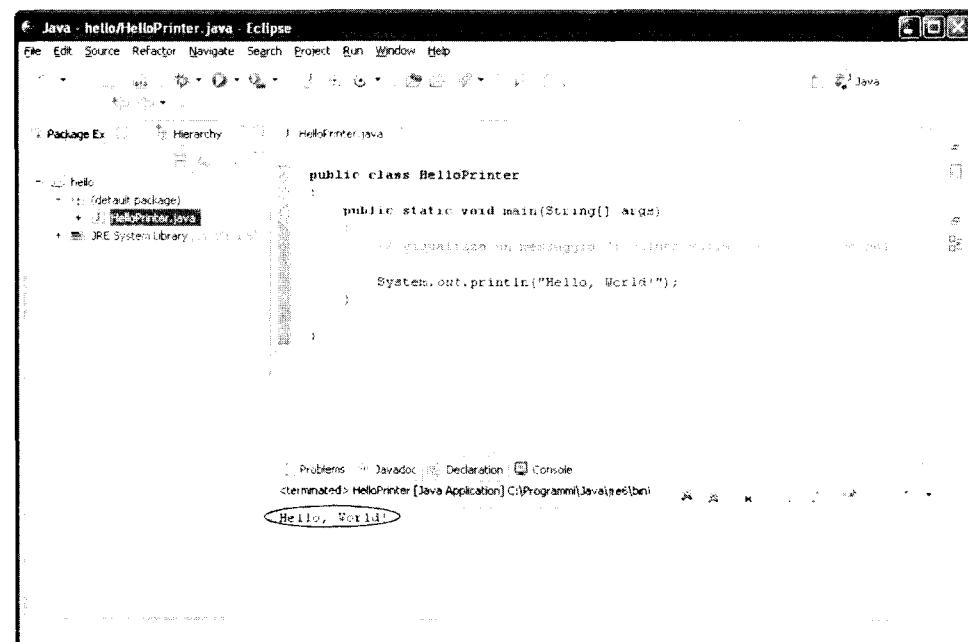
**Figura 8**

Esecuzione del programma HelloPrinter in una finestra di terminale

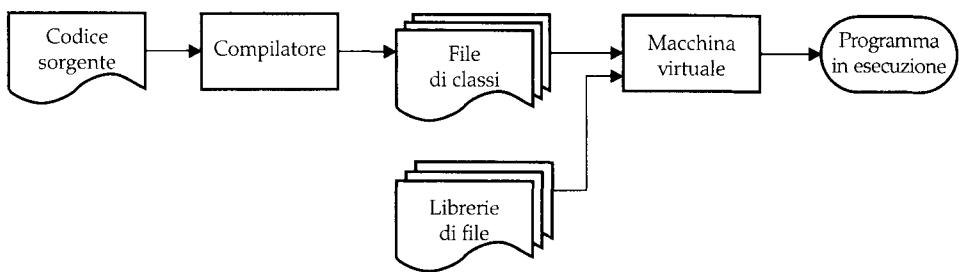
```
C:\>java>javac HelloPrinter.java
C:\>java>java HelloPrinter
Hello, World!
C:\>
```

**Figura 9**

Esecuzione del programma HelloPrinter in un ambiente di sviluppo integrato



**Figura 10**  
Dal codice sorgente al programma in esecuzione



l'estensione `.class`. Per esempio, le istruzioni per la macchina virtuale relative al programma `HelloPrinter` sono memorizzate nel file `HelloPrinter.class`. Si noti che, se il compilatore trova errori nel programma, non genera il corrispondente file di classe.

Il file di classe contiene la semplice traduzione delle istruzioni che avete scritto, che non bastano per eseguire effettivamente il programma: per visualizzare un messaggio in una finestra, sono necessarie alcune attività di basso livello. Gli autori delle classi `System` e `PrintStream` (che dichiara l'oggetto `out` e il metodo `println`) hanno implementato tutte le azioni necessarie e inserito i file di classe richiesti in una *libreria*, che è una raccolta di codice, programmato e tradotto da qualcun altro, pronto per l'utilizzo nel vostro programma.

La macchina virtuale Java carica le istruzioni del programma che avete scritto, avvia la sua esecuzione e carica i file di libreria che si rendono necessari.

I passaggi per compilare ed eseguire il vostro programma sono schematizzati nella Figura 10.

- Macchina virtuale Java carica
- esecuzione di un programma da file
- di classi e da file di libreria.



## Auto-valutazione

13. Potete usare un *word processor* (“elaboratore di testi”) per scrivere programmi Java?
14. Cosa vi aspettate di vedere caricando un file di classe all’interno di un editor di testo?



## Consigli per la produttività 1.1

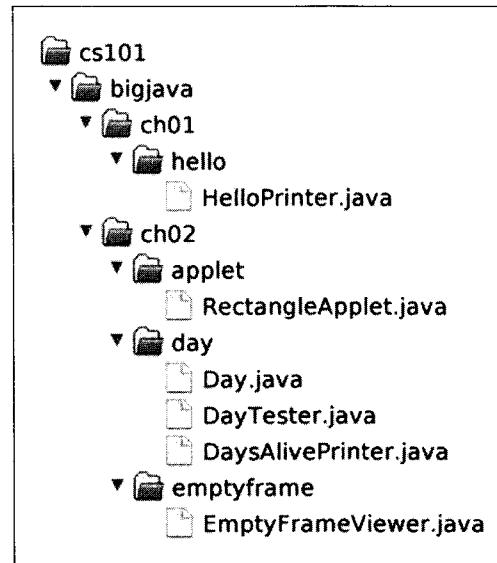
### Conoscere il *File System*

Negli ultimi anni l’utilizzo dei computer è stato molto semplificato, sia per applicazioni d’ufficio che per utilizzi personali, e oggi molti dettagli poco importanti rimangono nascosti ai normali utenti: ad esempio, molti memorizzano tutti i file necessari per il proprio lavoro all’interno di una cartella “predefinita” (con nomi quali “Documenti” o “Home”) e sono assai felici di poter ignorare i dettagli relativi all’organizzazione delle cartelle del proprio computer, organizzazione che viene chiamata *file system*.

Per il vostro lavoro di programmazione, invece, dovete acquisire consapevolezza del fatto che i file siano memorizzati in cartelle (*folder* o *directory*) e che tali contenitori possono essere *annidati*, l’uno dentro l’altro: questo significa che una cartella può contenere, oltre a file, anche altre cartelle, le quali, a loro volta, possono contenere file o altre cartelle, come si può vedere in Figura 11.

**Figura 11**

Cartelle annidate a costituire una gerarchia



Dovete sapere come strutturare e organizzare i dati che create e dovete anche saper rintracciare i vostri file per visualizzarne il contenuto.

Se non sapete farlo con disinvoltura, è decisamente meglio che dedichiate un po' di tempo all'apprendimento di questi concetti.



## Consigli per la produttività 1.2

### Adottare una strategia di *backup*

Trascorrerete molte ore scrivendo programmi Java e cercando di migliorarli, per cui i file sorgente che produrrette potrebbero essere importanti e dovreste averne cura, come siete soliti fare con altre cose importanti di vostra proprietà. È, quindi, essenziale che per i file presenti sul vostro computer sviluppiate con piena consapevolezza una strategia che ne garantisca la sicurezza.

I file sono più fragili dei documenti cartacei o di altri oggetti più tangibili: è facile cancellare un file accidentalmente e a volte i file vanno perduti a causa di un malfunzionamento del computer. A meno che non ne abbiate una copia, occorre digitare nuovamente il contenuto di un file perduto, ma probabilmente non lo ricorderete per intero ed è molto probabile che dobbiate perdere quasi lo stesso tempo che avevate impiegato a scriverlo e migliorarlo la prima volta. Ciò costituisce una perdita di tempo e può farvi fallire una scadenza, per cui è cruciale saper salvaguardare i propri file e prendere l'abitudine di farlo *prima* che sia troppo tardi. Occorre fare copie di sicurezza (o di *backup*) dei propri file, mettendoli in salvo in una diversa cartella, in una chiavetta o da qualche parte nella rete Internet.

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.

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

- *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 tre salvataggi recenti: anche se uno sarà difettoso, potrete usare uno dei rimanenti.
- *Fate il backup dei soli file di codice Java.* Il compilatore traduce i file di codice Java in file che contengono codice macchina: non è necessario salvare questi ultimi, dal momento che li potrete ricreare facilmente eseguendo di nuovo la compilazione. Concentratevi sui file che contengono il frutto dei vostri sforzi: in questo modo, i vostri supporti di backup non si riempiranno di cose inutili.
- *Fate attenzione alla destinazione del backup.* L'azione di backup consiste nel copiare file da un posto a un altro: è importante farlo nella direzione corretta, copiando nel supporto di backup i file che volete salvaguardare. 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 felice: fate un respiro profondo e riflettete sul processo di recupero prima di iniziare. Non è raro che un utente in preda all'agitazione distrugga il backup, nel tentativo di ripristinare un file danneggiato.

## 1.7 Errori

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

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

**Il codice è sintassi è una violazione delle regole del linguaggio e la 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 dal compilatore, ma potrebbe assomigliare a “*Cannot find symbol ou*” (“non sono in grado di trovare il simbolo `ou`”). Si tratta di un *errore in fase di compilazione* o *errore di sintassi*: in base alle regole del linguaggio, c'è qualcosa di sbagliato e il compilatore lo trova. Quando il compilatore individua errori, si rifiuta di tradurre il programma in istruzioni per la macchina virtuale Java e, di conseguenza, non avete un programma da eseguire: dovete 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.

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, per permettervi di sistemarli tutti in una volta sola.

Talvolta, tuttavia, un errore lo porta fuori strada. Pensate, ad esempio, a quello che succede se vi dimenticate la virgolette attorno a una stringa, scrivendo: `System.out.println(Hello, World!)`. Il compilatore non protesterà per le virgolette mancanti, bensì

segnerà “Cannot find symbol Hello”: sta a voi capire che, per correggere l’errore, dovete racchiudere la stringa tra virgolette.

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

```
Hello, Word!
```

Questo è un *errore in fase di esecuzione* o *errore logico*: il programma è sintatticamente corretto e fa qualcosa, ma non quello che ci si aspetta.

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 sovrumana 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. Il collaudo dei programmi è un argomento importante, che incontrerete molte volte in questo libro. Un altro aspetto importante, per una buona qualità del lavoro, è la *programmazione difensiva*, che significa strutturare i programmi e i processi di sviluppo in modo tale che un errore, situato in punto del programma, non innescchi esiti disastrosi.

Gli esempi di errori visti finora non erano difficili da diagnosticare e da correggere, ma, quando imparerete tecniche di programmazione più sofisticate, parallelamente vi saranno anche più occasioni di errore. È molto spiacevole che sia così difficile trovare tutti gli errori di un programma: anche vedendo che un programma produce un comportamento difettoso, potrebbe non essere affatto ovvio capire quale parte del programma lo causa e come sia possibile rimediare. Strumenti software specializzati, chiamati *debugger*, permettono di analizzare un programma per trovare i *bug*, gli errori logici. Nel Capitolo 6 imparerete a usare un debugger in modo efficace.

Notate che tutti questi errori sono di tipo diverso da quelli che di solito compaiono nelle operazioni di calcolo. Se sommate una colonna di numeri, potete dimenticare un segno meno o tralasciare un riporto, magari perché siete stanchi o annoiati. I computer non fanno questo tipo di errori.

In questo libro presenteremo una strategia di gestione degli errori in tre fasi. Innanzitutto vedrete gli errori più frequenti e come evitarli, poi apprenderete strategie di programmazione difensiva, per ridurre al minimo tanto la possibilità di fare errori quanto i loro effetti, e, infine, imparerete strategie di collaudo, per risalire agli errori superstiti.

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



## Auto-valutazione

15. Immaginate di omettere i caratteri `//` nel programma `HelloPrinter.java`, senza però eliminare la parte restante del commento. Otterrete un errore di sintassi o un errore logico?
16. Usando 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 da tastiera o con il mouse). Si tratta di un comportamento catalogabile come errore di sintassi o come errore logico?
17. Perché non potete collaudare un programma per scoprire errori logici se presenta errori di sintassi?



## Errori comuni 1.2

### Errori di ortografia

Se sbagliate 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 compilatore compilerà il metodo `Main` senza protestare ma, quando la macchina virtuale Java eseguirà il file compilato, si lamentera dell’assenza del metodo `main` e rifiuterà di eseguire il programma. Naturalmente, il messaggio “missing main method” (*il metodo main è assente*) dovrebbe fornirvi un indizio per cercare l’errore.

Se un messaggio di errore sembra indicare che il compilatore sia fuori strada, è il caso di controllare l’ortografia e le lettere maiuscole e minuscole. Se sbagliate a scrivere il nome di un simbolo (per esempio, `ou` al posto di `out`), il compilatore emetterà il messaggio d’errore “Cannot find symbol ou”: questo messaggio è, solitamente, un buon indizio di un possibile errore di ortografia.

## 1.8 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, consideriamo il processo di pianificazione che precede la realizzazione (detta anche *implementazione*) di un programma.

Può darsi che vi sia capitato di vedere annunci pubblicitari che vi 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 con i roller 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) 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.

Esaminiamo, ora, il seguente problema relativo a un investimento finanziario:

Depositate 10 000 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, in questo modo:

Anno	Saldo
0	10 000
1	$10\ 000.00 \times 1.05 = 10\ 500.00$
2	$10\ 500.00 \times 1.05 = 11\ 025.00$
3	$11\ 025.00 \times 1.05 = 11\ 576.25$
4	$11\ 576.25 \times 1.05 = 12\ 155.06$

Basta proseguire finché il saldo non è almeno pari a 20 000 dollari: a quel punto, la risposta è l'ultimo numero che avete 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	Saldo
0	10 000

Ripetere i passi seguenti finché il saldo è minore di 20 000.

Aggiungere 1 al valore dell'anno.

Moltiplicare il valore del saldo per 1.05 (incrementandolo, quindi, del 5 per cento).

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

Riportare il valore finale dell'anno come risposta.

Il pseudocodice è una descrizione informale di una sequenza di passi che portano alla soluzione di un problema.

Come è evidente, 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*.

Non esistono requisiti precisi e 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, usate enunciati come questi:

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

oppure

Moltiplicare il valore del saldo per 1.05.

oppure

Eliminare dalla parola il primo e l'ultimo carattere.

- Descrivete in questo modo decisioni e ripetizioni:

Se costo totale 1 < costo totale 2

oppure

Finché il saldo è minore di 20 000

oppure

Per ogni immagine presente nella sequenza

Usate i rientri verso destra del testo (“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.

- Specificate i risultati con enunciati simili a:

**Scegliere automobile1.**

**Riportare il valore finale dell’anno come risposta.**

Non è importante usare esattamente le stesse parole usate qui. Ciò che importa è che lo pseudocodice descriva una sequenza di passi

- Che non sia ambigua
- Che sia eseguibile
- Che termini 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 o all’inventiva. Un passo è, poi, *eseguibile* quando può essere effettivamente portato a termine. Se in un passo si chiedesse di usare il tasso di interesse reale dei prossimi anni, invece di un tasso di interesse prefissato e pari al 5 percento all’anno, esso non sarebbe eseguibile, perché nessuno ha oggi la possibilità di conoscere i tassi di interesse futuri. Infine, una sequenza di passi termina in un tempo finito se, prima o poi, ha certamente termine. Nel nostro esempio, bisogna fare un piccolo ragionamento per dimostrare che la sequenza di passi non verrà ripetuta all’infinito: ad ogni passo il saldo aumenta di almeno 500 dollari, per cui raggiungerà certamente il valore di 20 000 dollari.

Una sequenza di passi che non sia ambigua, che sia eseguibile e che termini in un tempo finito prende il nome di *algoritmo*. Dato che abbiamo individuato un algoritmo che risolve il nostro problema finanziario, possiamo risolverlo anche programmando un calcolatore: l’esistenza di un algoritmo è prerequisito essenziale per la programmazione. Prima di iniziare a programmare, dovete scoprire e descrivere un algoritmo adeguato al compito che volete risolvere (Figura 12).



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

## Auto-valutazione

- Nell’ipotesi che il tasso di interesse sia il 20 per cento, quanti anni servono per il raddoppio dell’investimento?
- 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.

**Figura 12**  
Procedimento per lo sviluppo di un programma



## Consigli pratici 1.1

### Sviluppo e descrizione di un algoritmo

Questa è la prima sezione di tipo “Consigli pratici” (*How To*) che trovate in questo libro: descrivono passo per 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, dovete sviluppare un algoritmo, cioè un metodo che consenta di giungere alla soluzione di uno specifico problema. Descrivete l’algoritmo mediante pseudocodice: un sequenza di passi ben precisi, espressi in linguaggio naturale.

Considerate, ad esempio, questo problema: dovete scegliere, per l’acquisto, tra due automobili, una delle quali è più efficiente dell’altra in merito al consumo di carburante, ma costa anche di più. 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 15 000 miglia all’anno, con un prezzo del carburante di \$ 4 al gallone. Pagate 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.

#### 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, possiamo 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 per carburante.

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

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

#### Fase 3 Descrivere ciascun sottoproblema mediante pseudocodice

Nella descrizione precedente, elencate i vari passi 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 percorse all'anno/efficienza

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

costo di esercizio =  $10 \times$  spesa annuale per carburante

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 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

$$\text{consumo annuale} = \text{miglia percorse all'anno}/\text{efficienza} = 15000/50 = 300$$

$$\text{spesa annuale per carburante} = \text{prezzo al gallone} \times \text{consumo annuale} = 4 \times 300 = 1200$$

$$\text{costo di esercizio} = 10 \times \text{spesa annuale per carburante} = 10 \times 1200 = 12000$$

$$\text{costo totale} = \text{prezzo} + \text{costo di esercizio} = 25000 + 12000 = 37000$$

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



## Esempi completi 1.1

### Sviluppo di un algoritmo per posare piastrelle su un pavimento

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

Come si posa una riga? Iniziate con 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 riga conterrà un numero di piastrelle pari a  $\text{larghezza}/4$ .

#### Fase 3 Descrivere ciascun sottoproblema mediante pseudocodice

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

Posate una piastrella nera nell'angolo nord-occidentale.

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

Ripetete questo passo  $((\text{larghezza}/4) - 1)$  volte:

Posate una piastrella a est di quella posata precedentemente.

Se era bianca, prendetene una nera, altrimenti, prendetene una bianca.

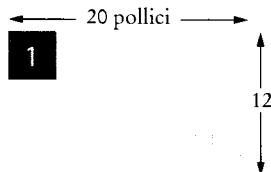
Identificate la piastrella iniziale della riga che avete appena posato.

Se a sud c'è spazio, posate là una piastrella di colore diverso.

**Fase 4** Collaudare lo pseudocodice in 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-occidentale.



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



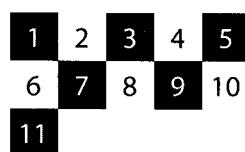
Verso sud c'è ancora spazio. Individuate la piastrella che si trova all'inizio della riga appena completata: è nera, quindi posate una piastrella bianca a sud di essa.



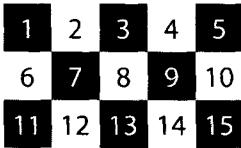
Completate la riga.



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



Completate la riga.



A questo punto il pavimento è completamente ricoperto: avete finito.

## Riepilogo degli obiettivi di apprendimento

### Definizione di “programma per calcolatore” e di “programmazione”

- Un computer, per svolgere compiti specifici, deve essere programmato: compiti diversi richiedono programmi diversi.
- Un programma per computer esegue una sequenza di istruzioni elementari a velocità molto elevata.
- Un programma contiene le sequenze di istruzioni necessarie a raggiungere tutti i suoi obiettivi.

### I componenti di un computer

- L’unità centrale di elaborazione (CPU, *central processing unit*) costituisce il cuore del computer.
- I dati e i programmi sono memorizzati nella memoria principale (semplicemente, memoria) e nella memoria secondaria (ad esempio, un disco rigido).
- La CPU legge dalla memoria le istruzioni, che la fanno comunicare con la memoria principale, la memoria secondaria e i dispositivi periferici.

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

- In generale, il codice macchina è specifico di ciascuna CPU, ma l’insieme delle istruzioni della JVM può essere eseguito da CPU diverse.
- Poiché le istruzioni macchina sono codificate come numeri, è difficile scrivere programmi in codice macchina.
- I linguaggi ad alto livello permettono la descrizioni dei compiti da svolgere a un livello concettuale più elevato rispetto al codice macchina.
- Un compilatore traduce in codice macchina programmi scritti in un linguaggio di alto livello.

### La storia del linguaggio di programmazione Java e dei principi su cui si basa

- Java venne originariamente progettato per programmare 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.
- 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.

### I blocchi costitutivi di un semplice programma e l’invocazione di un metodo

- Le classi sono i blocchi fondamentali per la costruzione di programmi Java.
- In ogni applicazione Java esiste una classe che ha il metodo `main`. Quando l’applicazione viene eseguita, vengono eseguite le istruzioni del metodo `main`.

- Usate i commenti per aiutare i lettori umani a capire il vostro programma.
- Per invocare un metodo si specifica un oggetto, il nome del metodo e i suoi parametri.
- Una stringa è una sequenza di caratteri racchiusa tra virgolette.

### L'ambiente di programmazione per scrivere ed eseguire programmi 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 (ad esempio, un programma Java).
- Java distingue tra maiuscolo e minuscolo.
- Impaginate il vostro programma in modo da agevolarne la lettura.
- Il compilatore Java traduce il codice sorgente in file di classi, che contengono istruzioni per la macchina virtuale Java.
- La macchina virtuale Java carica le istruzioni di un programma da file di classi e da file di libreria.
- Prima che si verifichi un disastro irrecuperabile, pianificate una strategia di backup del vostro lavoro.

### Catalogazione degli errori presenti in un programma: 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

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

## Classi, oggetti e metodi presentati nel capitolo

Ecco l'elenco di classi, metodi, variabili statiche e costanti presentati in questo capitolo.

<pre>java.io.PrintStream     print     println</pre>	<pre>java.lang.System     out</pre>
--	-------------------------------------

## Esercizi di ripasso

- \* **Esercizio R1.1.** Spiegate la differenza che intercorre fra usare un programma al calcolatore e programmare un computer.
- \* **Esercizio R1.2.** Che cosa distingue un computer da un comune elettrodomestico?
- \*\* **Esercizio R1.3.** Descrivete *esattamente*, passo dopo passo, le azioni che intraprendereste per predisporre una copia di salvataggio (cioè per "fare backup") del programma `HelloPrinter.java`, dopo averlo digitato.

- \*\* **Esercizio R1.4.** Nel vostro computer personale, oppure in quello del laboratorio, trovate la posizione esatta (nome della cartella o directory) di:
  - a. Il file di esempio `HelloPrinter.java`, che avete scritto con l'editor.
  - b. L'esecutore della macchina virtuale Java, `java.exe` oppure `java`.
  - c. Il file `rt.jar`, che contiene la libreria di esecuzione (*run-time library*).
- \* **Esercizio R1.5.** In che modo si scoprono gli errori di sintassi? Come si scoprono gli errori logici?
- \*\* **Esercizio R1.6.** Scrivete tre versioni del programma `HelloPrinter.java`, con errori di sintassi diversi. Scrivetene una versione con un errore logico.
- \*\*\* **Esercizio R1.7.** Cosa visualizzano gli enunciati seguenti? Non cercate di indovinare: scrivete ed eseguite i programmi per scoprirlo.
  - a. `System.out.println("3 + 4");`
  - b. `System.out.println(3 + 4);`
  - c. `System.out.println(3 + "4");`
- \*\* **Esercizio R1.8.** Scrivete un algoritmo per rispondere a questa domanda: un conto bancario contiene inizialmente \$ 10 000; gli interessi vengono calcolati mensilmente, al tasso annuo del 6 per cento (0.5 per cento al mese); ogni mese vengono prelevati \$ 500 per pagare le spese del convitto universitario; dopo quanti anni il conto è vuoto?
- \*\*\* **Esercizio R1.9.** Considerate di nuovo l'esercizio precedente e supponete che i valori che vi figurano (\$ 10 000, 6 per cento, \$ 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.
- \*\*\* **Esercizio R1.10.** 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 di ingresso l'altezza della casa e le sue due dimensioni orizzontali, 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).
- \*\* **Esercizio R1.11.** Volete decidere se andare al lavoro in automobile o in treno. Conoscete la distanza tra casa vostra e il luogo di lavoro, oltre all'efficienza della vostra automobile nel consumo di carburante (in miglia percorse per gallone consumato). Sapete anche il prezzo del biglietto ferroviario di sola andata. Ipotizzate che il carburante costi 4 dollari al gallone e che il costo di manutenzione dell'automobile sia pari a 5 centesimi di dollaro per ogni miglio percorso. Scrivete un algoritmo per decidere la modalità più economica per fare il pendolare.
- \*\* **Esercizio R1.12.** Volete calcolare la percentuale di utilizzo della vostra automobile per uso personale e, separatamente, per recarvi al lavoro. Conoscete la distanza tra casa vostra e il vostro luogo di lavoro e, per un certo periodo, avete registrato il valore iniziale e finale riportato dal contachilometri, oltre al numero di giorni lavorativi. Scrivete un algoritmo per risolvere questo problema.
- \* **Esercizio R1.13.** Nel problema descritto nei Consigli pratici 1.1, 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?



# 2

## Utilizzare gli 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*: per descrivere il comportamento di oggetti, in Java 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 classi.

## 2.1 Tipi

Prima di affrontare l'argomento principale di questo capitolo, dobbiamo presentare alcuni termini basilari per la programmazione, i concetti di tipo di dato, di variabile e di assegnazione.

**In tipo specifica un insieme di valori e le operazioni che possono essere compiute su di essi.**

Un programma per calcolatore elabora valori: numeri, stringhe e dati più articolati. In Java ogni valore è di un determinato *tipo*. Ad esempio, il numero 13 è di tipo `int` (un'abbreviazione di “integer”, *intero*), “Hello, World!” è di tipo `String` e l'oggetto `System.out` è di tipo `PrintStream`. Il tipo di valore indica cosa si può fare con il valore stesso: si può calcolare la somma o il prodotto di due numeri di tipo intero, mentre con oggetti di tipo `PrintStream` si può invocare il metodo `println`.

Java ha tipi distinti per *numeri interi* e per *numeri in virgola mobile*. I numeri interi sono privi di parte frazionaria, che invece può essere presente nei numeri rappresentati in virgola mobile. Ad esempio, 13 è un numero intero, mentre 1.3 è un numero in virgola mobile.

Il nome “virgola mobile” (*floating-point*) descrive la rappresentazione del numero all'interno del computer tramite la sequenza delle sue cifre significative, a cui si aggiunge l'indicazione della posizione del separatore decimale (“virgola” in italiano, “punto” nei paesi anglosassoni). Ad esempio, i numeri 13000, 1.3 e 0.00013 hanno le stesse cifre significative: 13. Quando un numero in virgola mobile viene moltiplicato o diviso per 10, si modifica solamente la posizione del separatore decimale, che diviene così “mobile” (in realtà, nei computer i numeri vengono rappresentati in base 2 e non in base 10, ma le considerazioni di principio non cambiano). Questa rappresentazione è in qualche modo simile alla notazione “scientifica”:  $1.3 \times 10^{-4}$ .

**Il tipo `double` caratterizza numeri in virgola mobile, che possono avere una parte frazionaria.**

Per elaborare numeri con parte frazionaria dovreste usare il tipo `double`, il cui nome significa “numero rappresentato in virgola mobile con *doppia precisione*”: un formato simile a quello dei numeri che vengono solitamente visualizzati da una calcolatrice tascabile, come 1.3 o  $-0.3333333333$ .

Un valore come 13 o 1.3 che figuri in un programma Java viene chiamato “letterale numerico” (*number literal*). Quando scrivete letterali numerici in Java non usate il punto per separare le migliaia (la virgola, nel sistema anglosassone), come si è invece soliti fare in ambito finanziario: ad esempio, 13.000 (“tredicimila”) va scritto `13000`. Per scrivere, in Java, numeri in notazione esponenziale, usate la notazione `E` invece di “ $\times 10^{\text{...}}$ ”: ad esempio,  $1.3 \times 10^{-4}$  si scrive `1.3E-4`. La Tabella 1 mostra come si scrivono, in Java, i letterali numerici di tipo intero e in virgola mobile.

Vi potreste a questo punto chiedere per quale motivo Java usi tipi diversi per i numeri interi e per i numeri con parte frazionaria: dopotutto, le calcolatrici tascabili non fanno questa distinzione e usano i numeri in virgola mobile per tutti i calcoli. I numeri interi hanno, però, diversi vantaggi rispetto ai numeri in virgola mobile: occupano meno spazio in memoria, vengono elaborati più velocemente e non danno luogo a errori di arrotondamento. Userete, quindi, il tipo `int` per quelle quantità che non possono mai avere una parte frazionaria, come la lunghezza di una stringa, e il tipo `double` per quelle quantità che possono, invece, avere una parte frazionaria, come la media dei vostri voti negli esami universitari.

Java mette a disposizione parecchi altri tipi numerici, che però vengono utilizzati di rado: ne parleremo nel Capitolo 4. In ogni caso, per la maggior parte dei programmi di questo libro i tipi `int` e `double` sono tutto ciò che vi serve per elaborare numeri.

**Tabella 1**

Letterali numerici 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 <b>double</b> .
1.0	double	Un numero intero con parte frazionaria .0 è di tipo <b>double</b> .
1E6	double	Un numero in notazione esponenziale: $1 \times 10^6$ , cioè 1 000 000. I numeri in notazione esponenziale sono sempre di tipo <b>double</b> .
2.96E-2	double	Esonente negativo: $2.96 \times 10^{-2} = 2.96 / 100 = 0.0296$ .
 100,000		<b>Errore:</b> non usare la virgola (né il punto) come separatore delle migliaia.
 3 1/2		<b>Errore:</b> non usare frazioni, soltanto la notazione decimale (qui, 3.5).

In Java, i tipi numerici sono tipi primitivi e i numeri non sono oggetti.

In Java i tipi numerici (`int`, `double` e gli altri, usati meno frequentemente) sono *tipi primitivi*. I numeri non sono oggetti e i tipi numerici non hanno metodi.

Tuttavia, è possibile elaborare numeri usando operatori quali `+` e `-`, all'interno di espressioni come `10 + n` oppure `n - 1`. Per moltiplicare due numeri si usa l'operatore `*`, ad esempio, `10 × n` si scrive `10 * n`.

Chiamiamo *espressione* una combinazione di variabili, letterali, operatori e/o metodi (che vedrete nel Paragrafo 2.4). Ecco un tipico esempio:

`x + y * 2`

Si possono combinare numeri in espressioni usando operatori aritmetici, quali `+`, `-` e `*`.

Come avviene in matematica, l'operatore `*` ha la precedenza rispetto all'operatore `+`, per cui `x + y * 2` rappresenta la somma tra `x` e `y * 2`. Se volete, invece, moltiplicare per 2 la somma di `x` e di `y`, usate le parentesi tonde:

`(x + y) * 2`

## Auto-valutazione

- Di che tipo sono i valori `0` e "0"?
- Quale tipo numerico usereste per memorizzare l'area di un cerchio?
- Perché l'espressione `13.println()` è errata?
- Scrivete un'espressione che calcoli la media tra i valori di `x` e di `y`.

## 2.2 Variabili

Spesso, all'interno di un programma, si vogliono memorizzare valori per poterli utilizzare in seguito. Per memorizzare un valore, bisogna conservarlo in una *variabile*, che è una porzione della memoria del computer caratterizzata da un *tipo*, un *nome* e un contenuto. Ad esempio, ecco come dichiariamo tre variabili:

```
String greeting = "Hello, World!";
PrintStream printer = System.out;
int width = 20;
```

Le variabili vengono usate per memorizzare valori che si vogliono utilizzare in seguito. Ogni variabile è caratterizzata da un tipo, un nome e un contenuto.

La prima variabile ha il nome `greeting`, può essere usata per memorizzare valori di tipo `String` e le viene assegnato il valore "Hello, World!". La seconda variabile memorizza un valore di tipo `PrintStream`, mentre la terza conserva al proprio interno un numero intero.

Le variabili possono, poi, essere utilizzate al posto dei valori che memorizzano:

```
printer.println(greeting); // equivale a System.out.println("Hello, World!");
printer.println(width); // equivale a System.out.println(20);
```

Quando dichiarate una variabile, dovete prendere due decisioni.

- Di che tipo deve essere?
- Che nome deve avere?

Il tipo di una variabile dipende dall'utilizzo che se ne intende fare: se dovete memorizzare una stringa, usate una variabile di tipo `String`; se, invece, vi serve un numero, scegliete il tipo `int` o `double`.

Memorizzare in una variabile un valore di un tipo che non corrisponde al tipo dichiarato per la variabile è un errore, come, ad esempio, questo:

```
String greeting = 20; // ERRORE: i tipi non corrispondono
```

Non si può usare una variabile di tipo `String` per memorizzare un numero intero. Il compilatore verifica la corrispondenza fra i tipi per proteggervi da questi errori.

**Tabella 2**

Dichiarazione di variabili in Java

Nome della variabile	Commento
<code>int width = 10;</code>	Dichiara una variabile intera e le assegna il valore iniziale 10.
<code>int area = width * height;</code>	Il valore iniziale può dipendere da altre variabili (che, ovviamente, devono essere state dichiarate in precedenza)
 <code>height = 5;</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 2.3).
 <code>int height = "5";</code>	<b>Errore:</b> non si può usare una stringa come valore iniziale di una variabile intera.
<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.

Nel decidere il nome di una variabile è sempre meglio fare una scelta che descriva lo scopo per cui viene utilizzata la variabile stessa: il nome di variabile `greeting` è migliore del nome `g`.

Il nome di una variabile, di un metodo o di una classe si chiama *identificatore* e Java impone le seguenti regole:

- Gli identificatori possono essere composti di lettere, cifre, caratteri “dollaro” (\$) e segni di sottolineatura (\_), ma non possono iniziare con una cifra.
- Non si possono usare altri simboli, come ? o %, né spazi.
- Inoltre, le *parole riservate*, come `public`, non possono essere usate come identificatori; tali parole sono riservate in modo esclusivo allo speciale significato che hanno nel linguaggio Java.

Gli identificatori di variabili, metodi e classi sono composti di lettere, cifre e segni di sottolineatura.

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

Nel linguaggio Java queste regole sono ferree: se ne violate una, il compilatore segnalerà un errore. Vi sono, poi, alcune *convenzioni* che dovraste seguire, in modo che altri programmati possano leggere con facilità i vostri programmi:

- I nomi di variabili e di metodi dovrebbero iniziare con una lettera minuscola. Si possono usare alcune lettere maiuscole all'interno del nome, come in `farewellMessage`, ma l'utilizzo misto di lettere maiuscole e minuscole viene a volte chiamato ironicamente “a cammello”, perché le lettere maiuscole svettano come le sue gobbe.
- I nomi di classi dovrebbero iniziare con una lettera maiuscola. Ad esempio, `Greeting` sarebbe un nome adatto a una classe, ma non a una variabile.
- Nei vostri identificatori non dovreste usare il simbolo \$, che è destinato ai nomi generati in modo automatico da alcuni strumenti di sviluppo.

Se non rispettate queste convenzioni il compilatore non segnalerà alcun errore, ma confonderete altri programmati che leggeranno il vostro codice.

La Tabella 3 presenta esempi di nomi di variabile leciti e invalidi in Java.

**Tabella 3**

Nome di variabili in Java

Nome della variabile	Commento
<code>farewellMessage</code>	Quando il nome della variabile è composto da più parole si usa la forma “a cammello”.
<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.
 <code>Greeting</code>	<b>Attenzione:</b> nei nomi di variabile, maiuscole e minuscole sono diverse. Questo nome di variabile è diverso da <code>greeting</code> .
 <code>6pack</code>	<b>Errore:</b> un nome di variabile non può iniziare con una cifra.
 <code>farewell message</code>	<b>Errore:</b> un nome di variabile non può contenere spazi.
 <code>public</code>	<b>Errore:</b> non si può usare una parola riservata del linguaggio come nome di variabile.



## Auto-valutazione

5. Quali dei seguenti identificatori sono validi?  
`Greeting1`  
`g`  
`void`  
`101dalmatians`  
`Hello, World`  
`<greeting>`
6. Dichiarate una variabile adatta a memorizzare il vostro nome, usando un identificatore “a forma di cammello”.

## 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.



### Consigli per la qualità 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 *p<sub>1</sub>*. 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 un'ampiezza (*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 “ampiezza”, 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 L'operatore di assegnazione

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;`

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

```
width = 20;
```

L'assegnazione provoca la sostituzione del valore originario della variabile, come si può vedere nella Figura 1.

**Figura 1**

Assegnazione di un nuovo valore a una variabile

```
width = 
width = 
```

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

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

da luogo a un errore: nel momento in cui utilizzate una variabile a cui non sia mai stato assegnato un valore (rappresentata nella Figura 2), il compilatore segnalerà la presenza di una “variabile non inizializzata” (*uninitialized variable*).

**Figura 2**

Una variabile priva di valore iniziale

```
height = 
```

Non è stato assegnato  
alcun valore

Tutte le variabili devono essere inizializzate prima di essere utilizzate.

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

```
int height;
height = 30;
width = height; // così è corretto
```

Oppure, ancora meglio, si può fornire un valore iniziale alla variabile nel momento in cui la si dichiara:

```
int height = 30;
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;
```

Calcola il valore dell'espressione a destra

`width = 30`

Memorizza il valore nella variabile

`width = 40`

`width + 10`

40

40

**Figura 3**

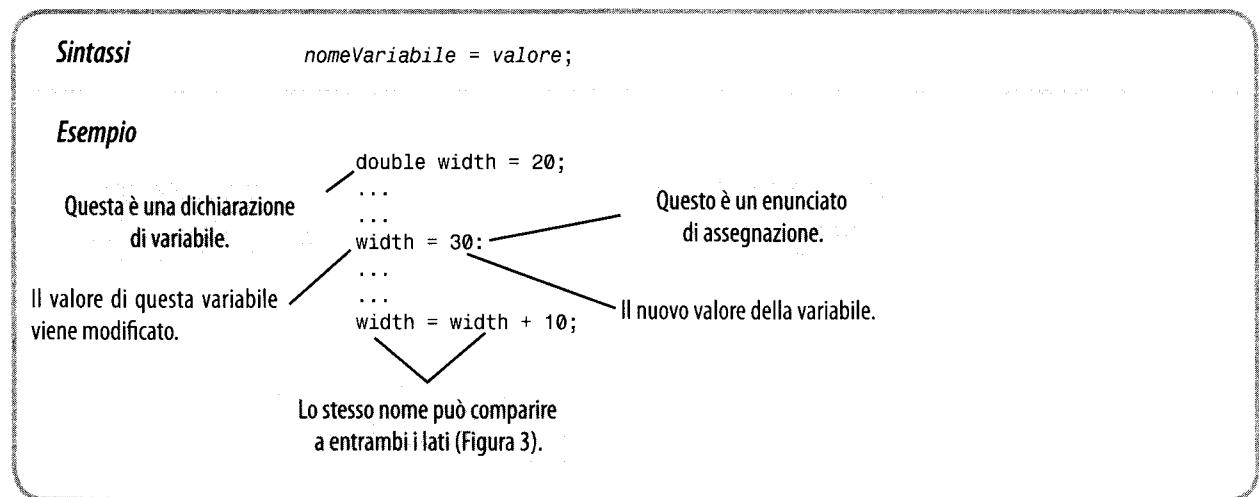
Esecuzione dell'enunciato  
`width = width + 10`

Ciò significa: "calcola il valore di `width + 10` e memorizzalo nella variabile `width`" (come rappresentato in Figura 3).

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 soluzioni.

## Sintassi di Java

## 2.2 Assegnazione



### Auto-valutazione

7. L'espressione `12 = 12` è valida nel linguaggio Java?
8. Come si fa a modificare la variabile `greeting` in modo che abbia il valore "Hello, Nina!" ?



### Errori comuni 2.1

#### Fare confusione tra gli 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.

## 2.4 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 accede ai dati  
di un oggetto.

Veniamo ora allo scopo principale di questo capitolo: comprendere meglio gli oggetti. Un *oggetto* è un valore che potete manipolare mediante l'invocazione dei suoi *metodi*. Un metodo è costituito da una sequenza di istruzioni che può accedere ai dati interni dell'oggetto: quando invocate 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 chiamate 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 della 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 4 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.

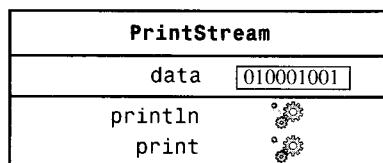
Nel Capitolo 1 abbiamo visto due oggetti:

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

Il tipo di un oggetto è una *classe*: 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.

**Figura 4**

Rappresentazione  
oggetto System.out



Una classe dichiara i metodi che possono essere applicati ai suoi oggetti.

Il metodo `println` può essere utilizzato con qualsiasi oggetto che appartenga alla classe `PrintStream` e `System.out` è uno di tali oggetti. Si possono ottenere altri oggetti della classe `PrintStream`: ad esempio, si può costruire un oggetto di tipo `PrintStream` per inviare a un file i dati prodotti da un programma (ma non ne parleremo fino al Capitolo 11).

Esattamente come la classe `PrintStream` fornisce, per i propri oggetti, metodi quali `println` e `print`, 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 n = greeting.length();
```

assegna a `n` il numero di caratteri presenti nella oggetto "Hello, World!" di tipo `String`. Dopo l'esecuzione delle istruzioni del metodo `length`, a `n` 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 all'interno delle parentesi tonde dati da elaborare (dati "in ingresso"), però fornisce un valore "in uscita", il conteggio dei caratteri.

Nel paragrafo successivo vedrete con maggiore dettaglio come si possano fornire dati in ingresso ai metodi e ottenere dati in uscita.

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`.

Quando applicate un metodo a un oggetto, dovete essere certi che il metodo sia dichiarato nella classe corrispondente. Ad esempio, questa invocazione è errata:

```
System.out.length(); // Questa invocazione di metodo è errata
```

perché la classe `PrintStream`, a cui appartiene `System.out`, non ha un metodo `length`.

Riassumiamo. In Java, *ogni oggetto appartiene a una classe e una classe dichiara i metodi per i suoi oggetti*. Ad esempio, la classe `String` definisce i metodi `length` e `toUpperCase` (oltre ad altri metodi, molti dei quali saranno presentati nel Capitolo 4). I metodi di una classe costituiscono la sua *interfaccia pubblica*, 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 programmati 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 specificata dalla classe `String`.

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.

**Figura 5**

Rappresentazione di due oggetti di tipo String

String	
	data = Hello...
length	•••
toUpperCase	•••

String	
	data = Missi...
length	•••
toUpperCase	•••

A volte una classe dichiara due metodi aventi lo stesso nome e 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 valore intero. Diciamo, in questo caso, che il nome `println` è *sovraccarico* ("overloaded", nel senso di "carico di più significati"), dal momento che si riferisce a più metodi.



### Auto-valutazione

9. In che modo si può calcolare la lunghezza della stringa "Mississippi"?
10. In che modo si può visualizzare la versione maiuscola della stringa "Hello, World!"?
11. Si può invocare `river.println()`? Perché o perché no?

## 2.5 Parametri e valori restituiti dei metodi

I metodi sono elementi costitutivi fondamentali dei programmi Java: un programma fa qualcosa di utile invocando metodi. In questo paragrafo vedrete come fornire dati in ingresso a un metodo e come ottenerne in uscita il risultato prodotto.

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

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

La Figura 6 evidenzia il trasferimento del parametro al metodo.

Dal punto di vista tecnico, il parametro `greeting` è un *parametro esplicito* del metodo `println`. Un altro parametro nell'invocazione di un metodo è l'oggetto con cui si invoca il metodo stesso: si parla, in questo caso, di *parametro隐式*. Ad esempio, `System.out` è il parametro implicito nell'invocazione seguente

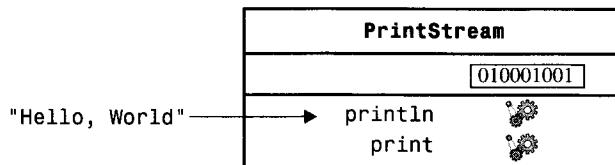
```
System.out.println(greeting);
```

Alcuni metodi hanno bisogno di più parametri esplicativi, 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

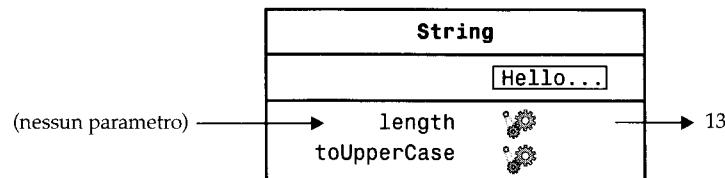
Il parametro è un dato fornito  
in ingresso a un metodo.

Il parametro implicito  
dell'invocazione di un metodo  
è l'oggetto con cui viene invocato  
il metodo stesso. Tutti gli altri  
parametri si dicono esplicativi.

**Figura 6**  
Passaggio di parametro  
al metodo println



**Figura 7**  
Invocazione del metodo  
length su un oggetto  
di tipo String



svolgimento del proprio compito, cioè per contare i caratteri presenti nella stringa, sono memorizzate nell'oggetto che ne costituisce il parametro implicito.

Il valore restituito da un metodo è il risultato da esso calcolato, da utilizzare nel codice che ha invocato il metodo stesso.

Il metodo `length` differisce dal metodo `println` anche sotto un altro aspetto: calcola un dato. In questa situazione, si dice che il metodo *restituisce un valore*, che in questo caso è il numero di caratteri contenuti nella stringa. Tale valore restituito può essere, ad esempio, memorizzato in una variabile:

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

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.

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

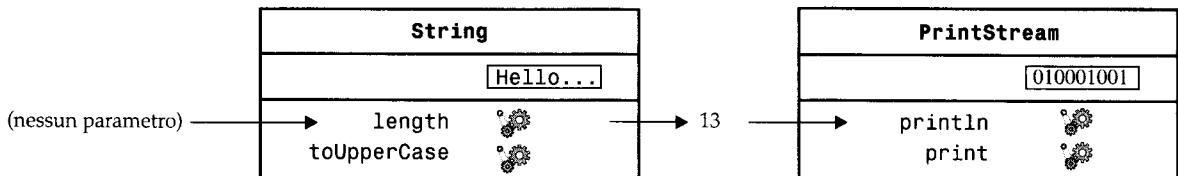
Analizziamo ora un'invocazione di metodo un po' più complessa: invochiamo il metodo `replace` della classe `String`. Il metodo `replace` esegue operazioni di ricerca e sostituzione, analogamente a quanto avviene in un *word processor*. Ad esempio, l'invocazione

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

**Figura 8**

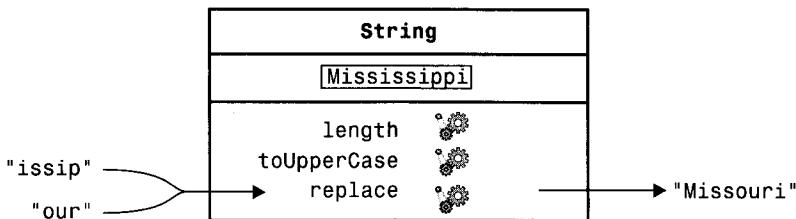
Il valore restituito da un metodo può essere utilizzato come parametro per un altro metodo

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



**Figura 9**

Invocazione del metodo  
replace



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

oppure passato come parametro a un altro metodo:

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

Come si vede nella Figura 9, questa invocazione di metodo ha

- un parametro implicito: la stringa "Mississippi"
- due parametri esplicativi: le stringhe "issipp" e "our"
- un valore restituito: la stringa "Missouri"

Quando in una classe si dichiara un metodo, vengono specificati i tipi dei parametri esplicativi 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 esplicativi e restituisce un valore di tipo **int** (per il momento considereremo “pubblici” tutti i metodi, mentre nel Capitolo 10 vedremo metodi con accesso più ristretto).

Il tipo del parametro implicito è la classe in cui è dichiarato il metodo, nel nostro caso **String**: ciò non viene menzionato nella definizione del metodo e proprio per questo si parla di parametro “implicito”.

Il metodo **replace** è invece dichiarato in questo modo:

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

Per invocare il metodo **replace** bisogna fornire due parametri esplicativi, **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 riservata **void**. Ad esempio, la classe **PrintStream** definisce il metodo **println** in questo modo:

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

## Auto-valutazione

- Quali sono i parametri impliciti, i parametri esplicativi e il valore restituito nell’invocazione di metodo **river.length()**?



13. Qual è il risultato dell'invocazione `river.replace("p", "s")?`
14. Qual è il risultato dell'invocazione  
`greeting.replace("World", "Dave").length()?`
15. Come viene dichiarato il metodo `toUpperCase` nella classe `String`?

## 2.6 Costruire oggetti

La maggior parte dei programmi Java opera su molti diversi oggetti. In questo paragrafo vedrete come costruire nuovi oggetti, per andare oltre il mero utilizzo di oggetti di tipo `String` e dell'oggetto `System.out`.

Per apprendere come si costruiscano oggetti, studiamo un'altra classe, la classe `Rectangle` della libreria di classi di Java. Gli oggetti di tipo `Rectangle` descrivono forme rettangolari (come potete vedere nella Figura 10), 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.

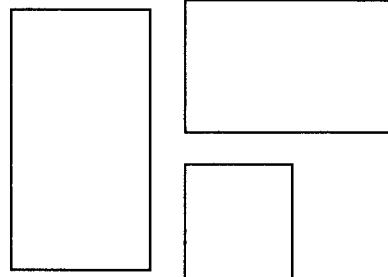
Notate che un oggetto di tipo `Rectangle` non è una forma rettangolare, ma un oggetto che contiene un insieme di numeri che *descrivono* il rettangolo (osservare la Figura 11). Ciascun rettangolo è descritto dalle coordinate `x` e `y` del suo vertice superiore sinistro, dalla sua larghezza (`width`) 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 specificare 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 potete costruire un rettangolo con le coordinate del vertice superiore sinistro uguali a (5, 10), con larghezza 20 e altezza 30:

**Figura 10**

Forme rettangolari



**Figura 11**

Oggetti di tipo `Rectangle`

**Rectangle**

<code>x =</code>	<input type="text" value="5"/>
<code>y =</code>	<input type="text" value="10"/>
<code>width =</code>	<input type="text" value="20"/>
<code>height =</code>	<input type="text" value="30"/>

**Rectangle**

<code>x =</code>	<input type="text" value="35"/>
<code>y =</code>	<input type="text" value="30"/>
<code>width =</code>	<input type="text" value="20"/>
<code>height =</code>	<input type="text" value="20"/>

**Rectangle**

<code>x =</code>	<input type="text" value="45"/>
<code>y =</code>	<input type="text" value="0"/>
<code>width =</code>	<input type="text" value="30"/>
<code>height =</code>	<input type="text" value="20"/>

```
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*. I quattro valori 5, 10, 20 e 30 rappresentano i *parametri di costruzione*.

L'espressione `new` fornisce un oggetto, che va memorizzato da qualche parte se lo si vuole utilizzare in seguito. 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).

### Auto-valutazione

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

## Sintassi di Java

### 2.3 Costruzione di oggetti

#### Sintassi

```
new NomeClasse(parametri)
```

#### Esempio

L'espressione `new` restituisce un oggetto.

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

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

Parametri di costruzione

```
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.

17. Il metodo `getWidth` restituisce la larghezza di un oggetto di tipo `Rectangle`. Cosa viene visualizzato dal seguente enunciato?

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



## Errori comuni 2.2

### 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
```

## 2.7 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*. Diversamente, un metodo che abbia lo scopo di modificare i dati interni di un oggetto viene chiamato *metodo modificatore*.

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` forniscono, 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();
```

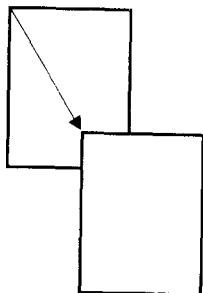
Esaminiamo ora un metodo modificatore. I programmi che manipolano rettangoli hanno spesso bisogno di spostarli, ad esempio per visualizzare animazioni. La classe `Rectangle` ha un metodo, chiamato `translate`, che serve proprio a questo: sposta un rettangolo di una certa quantità nelle direzioni `x`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

**Figura 12**

Uso del metodo `translate` per spostare un rettangolo



Questo metodo è un metodo modificatore perché modifica l'oggetto che ne costituisce il parametro隐式.



### Auto-valutazione

18. Il metodo `toUpperCase` della classe `String` è un metodo d'accesso o un metodo modificatore?
19. Quale invocazione del metodo `translate` è necessaria per spostare il rettangolo `box = new Rectangle(5, 10, 20, 30)` in modo che il suo vertice superiore sinistro si porti nel punto `(0, 0)`, origine delle coordinate?

## 2.8 La documentazione API

Le classi e i metodi della libreria di Java sono elencati nella *documentazione API* (Application Programming Interface, 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://java.sun.com/javase/7/docs/api/index.html>. La documentazione API illustra tutte le classi della libreria di Java, che sono migliaia (ne vedete un esempio nella Figura 13) e spesso sono estremamente specifiche, per cui soltanto poche sono di interesse generale per gli apprendisti programmatori.

Cercate nel riquadro di sinistra il collegamento `Rectangle`, possibilmente utilizzando la funzione di ricerca del vostro browser. Attivate tale collegamento e vedrete visualizzata nel riquadro di destra la documentazione di tutte le caratteristiche della classe `Rectangle`, come nella Figura 14.

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 Figura 15. Selezionando il nome di un metodo si giunge a una sua descrizione più approfondita, come mostrato nella Figura 16.

La descrizione dettagliata di un metodo contiene:

- l'azione svolta dal metodo
- i parametri ricevuti dal metodo

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

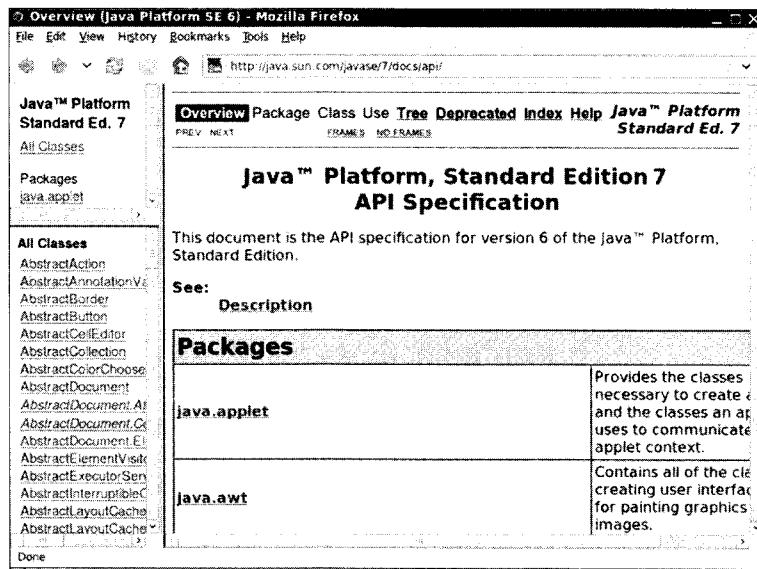
- il valore restituito dal metodo (oppure la parola riservata `void`, se il metodo non restituisce alcunché)

Come potete notare, la classe `Rectangle` ha parecchi metodi: anche se ciò può mettere inizialmente un po' in soggezione il programmatore inesperto, 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. Navigando nella documentazione API, troveremo un metodo `setSize`, avente queste

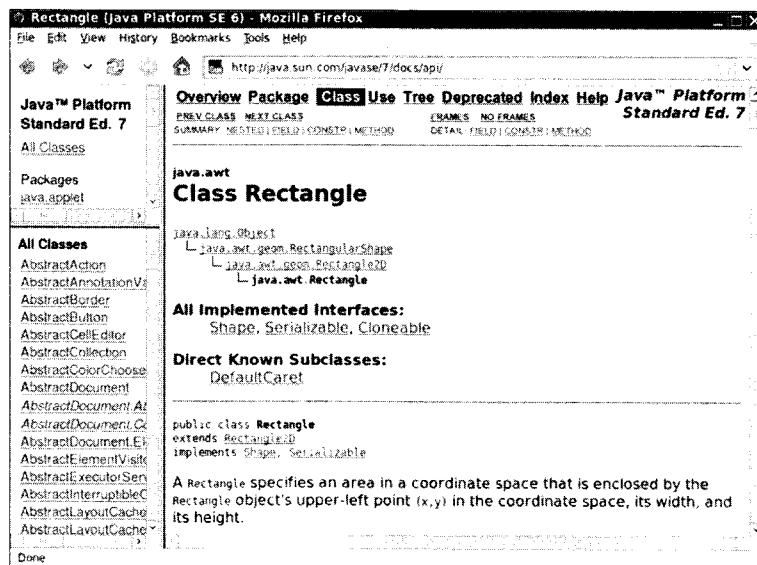
**Figura 13**

La documentazione API per la libreria standard di Java



**Figura 14**

La documentazione API per la classe Rectangle



descrizione: "Imposta la dimensione di questo `Rectangle` al valore specificato per la larghezza e l'altezza". Il metodo richiede due parametri, così descritti:

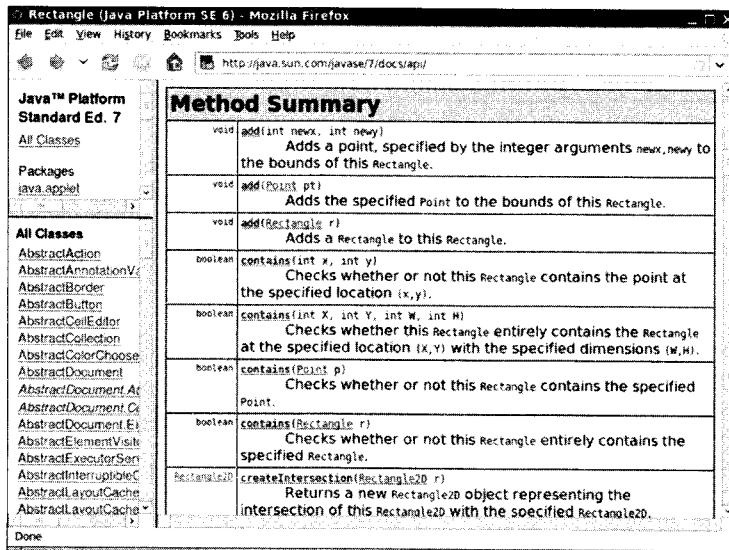
- `width` – la nuova larghezza di questo `Rectangle`
- `height` – la nuova altezza di questo `Rectangle`

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);
```

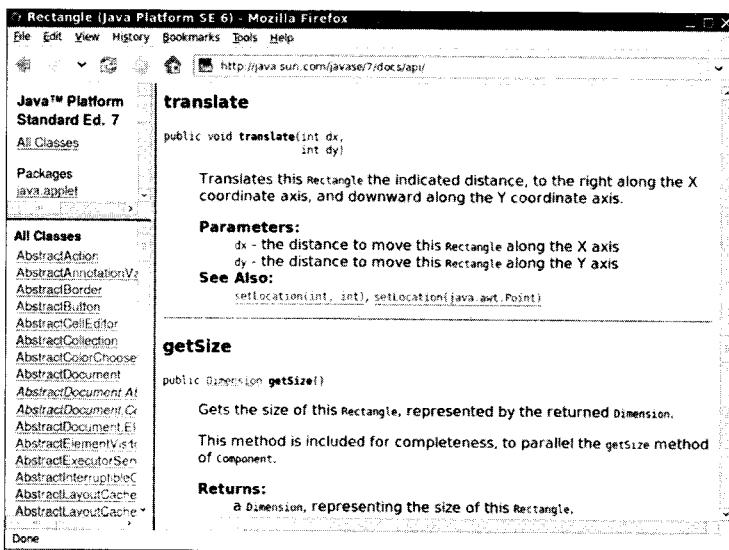
**Figura 15**

L'elenco riassuntivo dei metodi della classe `Rectangle`



**Figura 16**

La documentazione API del metodo `translate`



## Sintassi di Java

## 2.4 Importazione di una classe da un pacchetto

### Sintassi

```
import nomePacchetto.NomeClasse;
```

### Esempio

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

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

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 utilità 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 Figura 14, il nome del pacchetto, `java.awt`, è visibile proprio al di sopra del nome della classe.

Le classi Java sono raggruppate in pacchetti. Per utilizzare classi definite in pacchetti diversi da quelli standard si usa l'enunciato `import`.

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

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

Perché non dobbiamo 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

20. Consultate la documentazione API della classe `String`. Quale metodo usereste per ottenere la stringa “hello, world!” a partire da “Hello, World!”?
21. Consultate la descrizione del metodo `trim` nella documentazione API della classe `String`. Cosa si ottiene applicando il metodo `trim` alla stringa “ Hello, Space ! ” ? (Notate gli spazi presenti nella stringa)
22. La classe `Random` viene dichiarata nel pacchetto `java.util`. Cosa dovete fare per usare tale classe in un vostro programma?



### Consigli per la produttività 2.1

#### 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 dall'indirizzo <http://java.sun.com/javase/downloads/index.html>.

## 2.9 Realizzare un programma di collaudo

In questo paragrafo illustreremo le fasi necessarie per la realizzazione 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. Definire una classe di collaudo.
2. Definire in essa il metodo `main`.
3. Costruire uno o più oggetti all'interno del metodo `main`.
4. Applicare metodi agli oggetti.
5. Visualizzare i risultati delle invocazioni dei metodi.
6. Visualizzare i valori previsti da tali invocazioni.

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

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.

Questo è un aspetto molto importante: prima di eseguire un programma di collaudo, dovete meditare 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.

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

Nel nostro caso, il rettangolo viene costruito con l'angolo superiore sinistro nella posizione (5, 10), poi viene spostato nella direzione `x` di 15 pixel, 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 ch02/rectangle/MoveTester.java

```

import java.awt.Rectangle;

public class MoveTester
{
    public static void main(String[] args)
    {
        Rectangle box = new Rectangle(5, 10, 20, 30);

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

        // visualizza informazioni sul rettangolo traslato
        System.out.print("x: ");
        System.out.println(box.getX());
        System.out.println("Expected: 20");

        System.out.print("y: ");
        System.out.println(box.getY());
        System.out.println("Expected: 35");
    }
}

```

### Esecuzione del programma

```

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

```



### Auto-valutazione

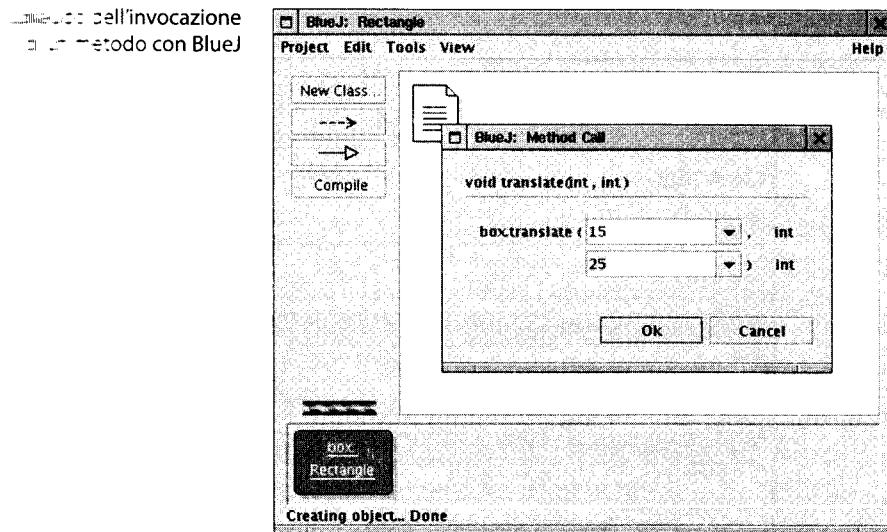
23. Se avessimo invocato `box.translate(25,15)` invece di `box.translate(15, 25)`, quali sarebbero stati i valori previsti?
24. Perché non c'è bisogno che il programma `MoveTester` visualizzi la larghezza e l'altezza del rettangolo?



### 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 dall'indirizzo <http://www.bluej.org>. Un altro eccellente ambiente per l'esplorazione interattiva di oggetti è Dr. Java (<http://drjava.sourceforge.net>).



## 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 `ch02/day` 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 correntemente 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. Tutti questi dettagli vengono gestiti dai meccanismi interni della classe `Day`.

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

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

Avete ora il compito di 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 sottocartella `ch02/day/api`: come potete vedere nella figura, l'enunciato seguente costruisce un oggetto di tipo `Day` a partire da tre valori, che rappresentano anno, mese e giorno.

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

The screenshot shows a Mozilla Firefox browser window displaying the JavaDoc for the `Day` class. The URL in the address bar is `file:///home/cay/book/tbg4/codexch02/day/api/index.html`. The page title is "Day - Mozilla Firefox".

**Package Class Tree Deprecated Index Help**

**Class Day**

`java.lang.Object`  
↳ `Day`

**public class Day**  
extends Object

**Constructor Summary**

<code>Day()</code>	Constructs a day object representing today's date.
<code>Day(int year, int month, int alive)</code>	Constructs a day with a given year, month, and day of the Julian/Gregorian calendar.

**Method Summary**

<code>Day addDays(int n)</code>	Returns a day that is a certain number of days away from this day.
<code>int daysFrom(Day other)</code>	Returns the number of days between this day and another day.
<code>int getDate()</code>	Returns the day of the month of this day.
<code>int getMonth()</code>	Returns the month of this day.
<code>int getYear()</code>	Returns the year of this day.
<code>String toString()</code>	Returns a string representation of the object.

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 vostro problema, a meno che non vogliate sostituire il valore 100 con altri valori, provando e riprovando fino ad ottenere la data odierna. Bisogna, invece, 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:

Return: the number of days that this day is away from the other  
(>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)”. Inoltre, nella documentazione con *questo* (`this`) si intende sempre il parametro implicito del metodo. Noi vogliamo un risultato positivo, quindi la forma corretta è la seconda.

Ecco, infine, il programma che risolve il nostro problema (lo trovate nella cartella `ch02/day`):

```
public class DaysAlivePrinter
{
    public static void main(String[] args)
    {
        Day jamesGoslingsBirthday = new Day(1955, 5, 19);
        Day today = new Day();
        System.out.print("Today: ");
        System.out.println(today.toString());
        int daysAlive = today.daysFrom(jamesGoslingsBirthday);
        System.out.print("Days alive: ");
        System.out.println(daysAlive);
    }
}
```

## Esecuzione del programma

```
Today: 2009-10-05
Days alive: 19863
```

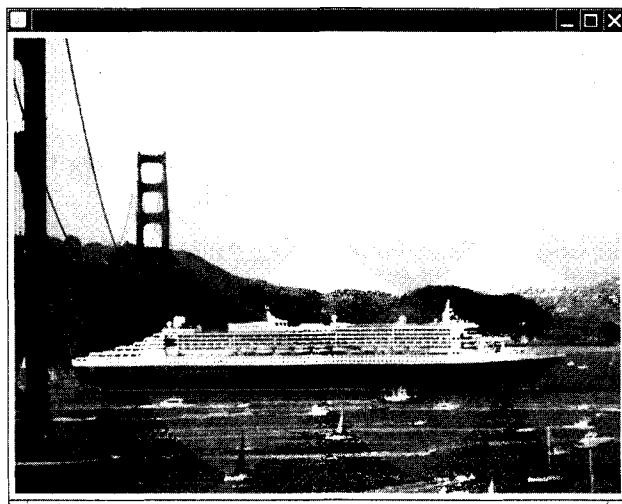


## Esempi completi 2.2

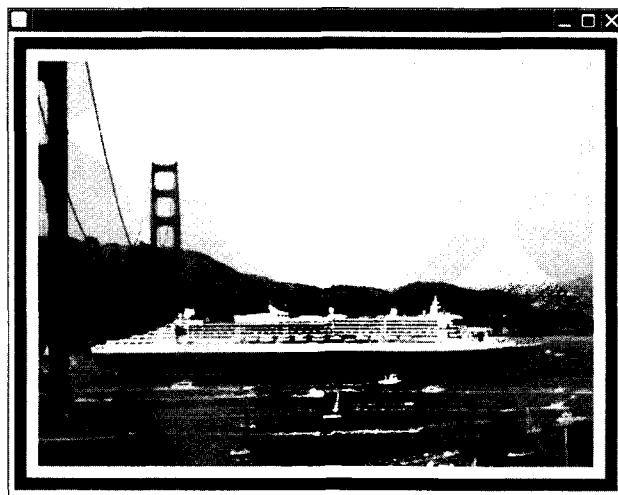
### Lavorare con immagini

La cartella `ch02/picture` nel pacchetto dei file scaricabili per questo libro 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:

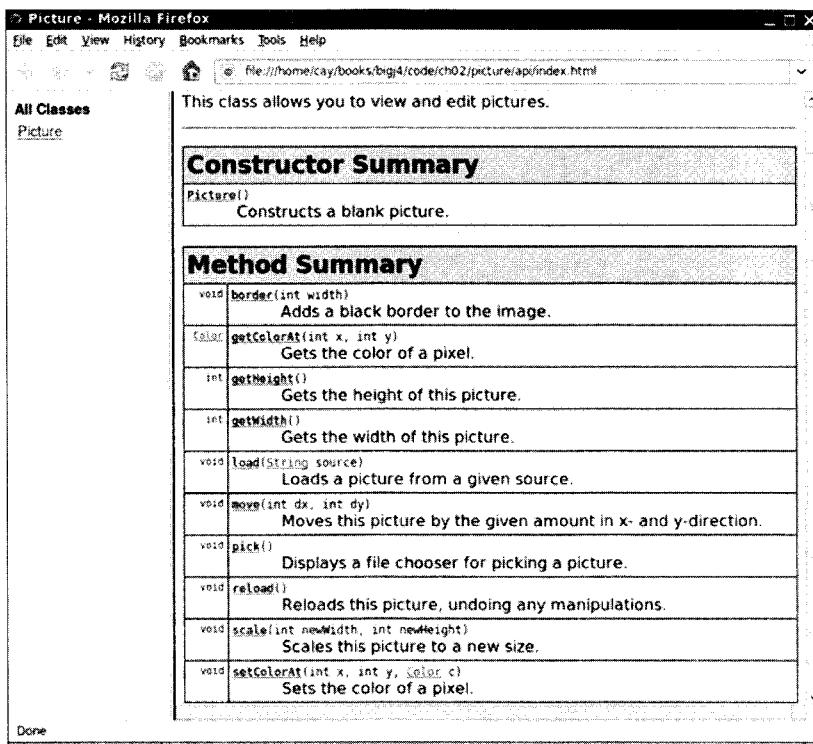
```
public class PictureDemo
{
    public static void main(String[] args)
    {
        Picture pic = new Picture();
        pic.load("queen-mary.png");
    }
}
```



Vogliamo ora 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 nella figura qui sotto.



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 ch02/picture/api.



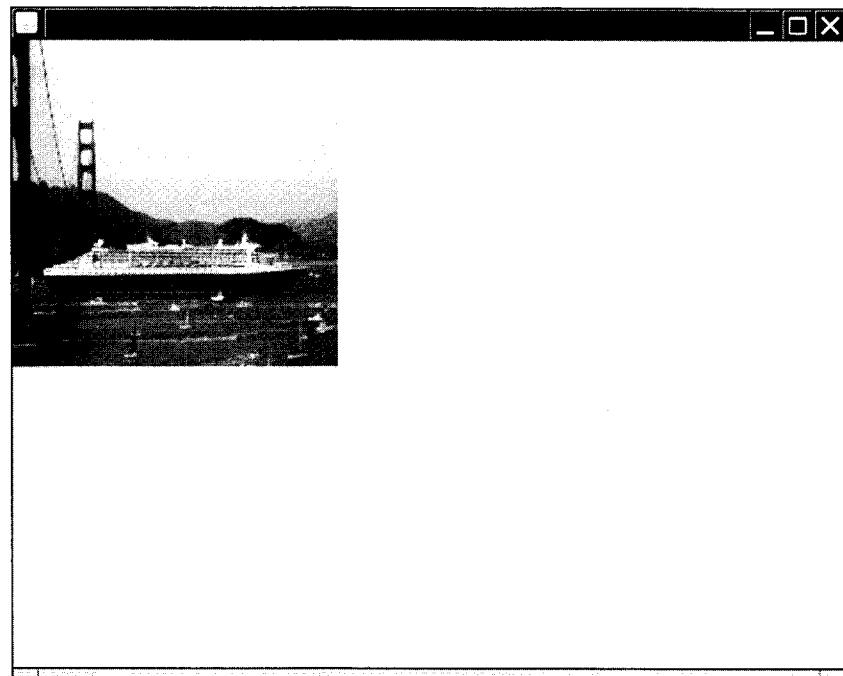
La classe contiene alcuni metodi che non servono al nostro scopo, ma due di essi ci saranno chiaramente utili:

```
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 appena 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 40 pixel inferiori ai valori originari, lasciando così 20 pixel liberi su ogni lato per i bordi.

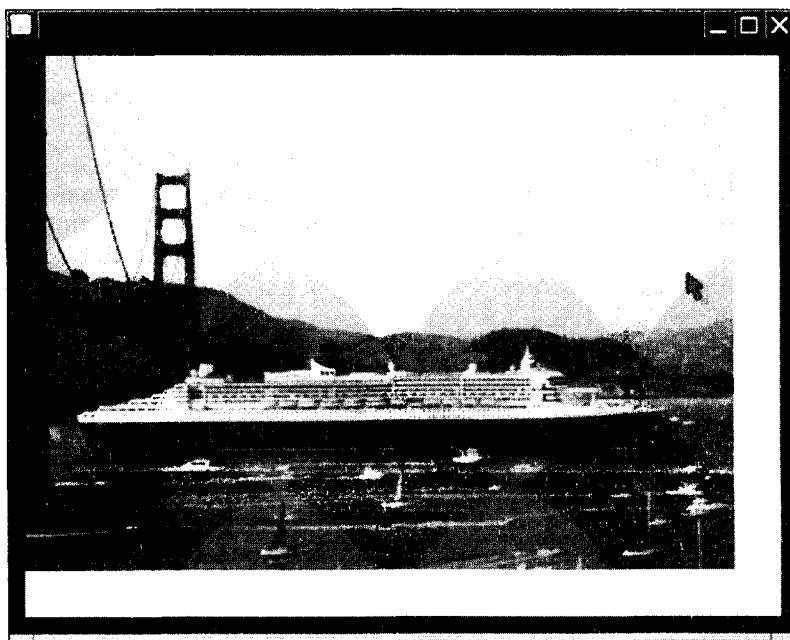
Consultando la documentazione API, troviamo i metodi che ci consentono di ispezionare i valori originari di larghezza e altezza dell'immagine, per cui scriveremo:

```
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:

```
public class BorderMaker
{
    public static void main(String[] args)
    {
        Picture pic = new Picture();
        pic.load("queen-mary.png");
        int newWidth = pic.getWidth() - 40;
        int newHeight = pic.getHeight() - 40;
        pic.scale(newWidth, newHeight);
        pic.move(20, 20);
        pic.border(10);
    }
}
```

Non 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.

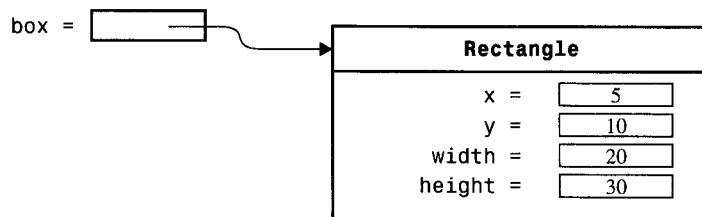
## 2.10 Riferimenti a oggetti

In Java, una variabile il cui tipo sia una classe non contiene, in realtà, un oggetto: memorizza al proprio interno soltanto la *posizione* dell'oggetto all'interno della memoria del computer. L'oggetto è memorizzato altrove, come si può vedere nella Figura 17.

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 17**

Una variabile oggetto contenente un riferimento a un oggetto



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

Più variabili oggetto possono fare riferimento al medesimo oggetto.

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`.

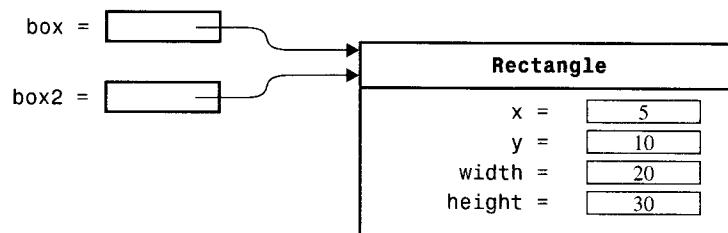
Occorre assolutamente ricordare che la variabile `box` *non contiene* l'oggetto, ma *fa solamente riferimento* a esso. Si possono anche avere due variabili oggetto che si riferiscono al medesimo oggetto:

```
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 18.

**Figura 18**

Due variabili oggetto che fanno riferimento al medesimo oggetto



**Figura 19**

Una variabile di tipo numerico memorizza un numero

`luckyNumber = 13`

Le variabili numeriche, al contrario, memorizzano veramente i numeri, per cui quando si dichiara

```
int luckyNumber = 13;
```

la variabile `luckyNumber` contiene effettivamente il numero 13, come si può vedere nella Figura 19, 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.

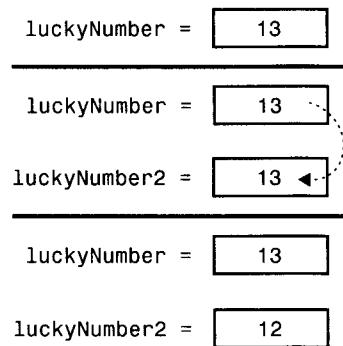
Quando si copia una variabile si apprezza bene la differenza tra variabili di tipo numerico e variabili oggetto. Quando si copia un numero, il numero originario e la sua copia sono valori tra loro indipendenti, mentre quando si copia un riferimento a un oggetto l'originale e la copia sono riferimenti 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 20:

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

**Figura 20**

Copertura di numeri



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

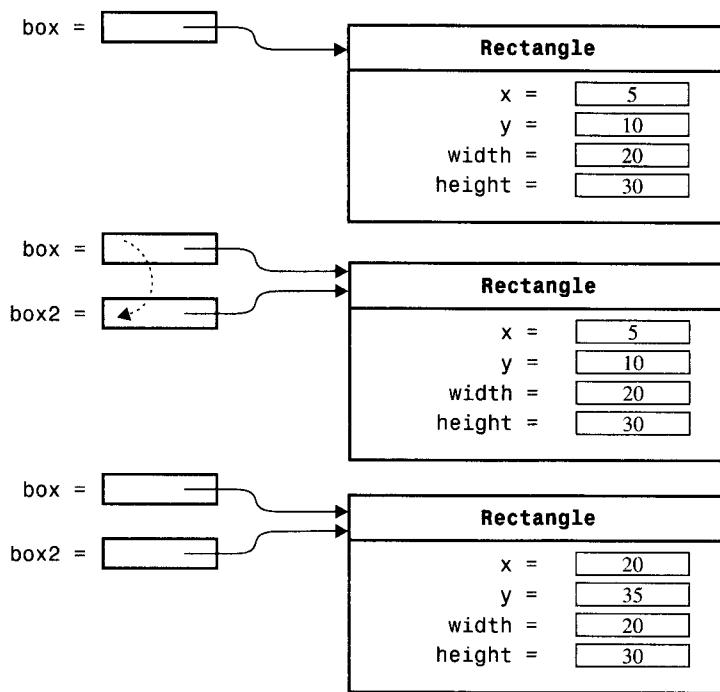
Considerate ora il codice, apparentemente simile, che effettua la copia di oggetti di tipo `Rectangle`, come rappresentato nella Figura 21:

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

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

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

**Figura 21**  
Copiatura di riferimenti a oggetti



Non c'è, però, bisogno che vi preoccupiate 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 (all'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

25. Che effetto ha l'assegnazione `greeting2 = greeting`?
26. Dopo aver invocato `greeting2.toUpperCase()`, cosa contengono `greeting` e `greeting2`?

## 2.11 Applicazioni grafiche e finestre

Questo è il primo di una serie di paragrafi che vi insegheranno a progettare *applicazioni grafiche*, cioè applicazioni che visualizzano disegni all'interno di una finestra e hanno un'attrattiva decisamente maggiore delle applicazioni per finestre di solo testo.

Un'applicazione grafica visualizza informazioni all'interno di una “finestra di tipo `frame`” (brevemente, “un frame”), cioè una finestra dotata di una barra di titolo, come mostrato nella Figura 22. In questo paragrafo vedrete come si visualizza un frame, mentre nel Paragrafo 3.9 imparerete a creare disegni al suo interno.

Per visualizzare un frame si costruisce un oggetto di tipo `JFrame`, se ne impostano le dimensioni e lo si rende visibile.



## Note di cronaca 2.1

### I mainframe: quando i dinosauri dominavano la terra

Quando, nei primi anni '50, 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 esisteva un mercato per non più di 50 apparecchiature di quel tipo, destinate a organizzazioni governative o militari e ad alcune delle più grandi aziende nazionali. Invece, vendettero circa 1500 esemplari del loro modello System 650 e iniziarono a produrre e vendere computer più potenti.

I cosiddetti computer *mainframe* degli anni '50, '60 e '70 erano enormi. Riempivano stanze intere, che bisognava climatizzare per proteggerne le delicate apparecchiature, come si vede in figura. Attualmente, grazie alle tecnologie di miniaturizzazione, anche i mainframe sono diventati più piccoli, però sono ancora molto costosi (mentre scriviamo, un tipico mainframe costa parecchi milioni di dollari).

Questi sistemi enormi e costosi ebbero un immediato successo appena apparvero sul mercato, perché prendevano il posto di molti uffici affollati da impiegati ancora più costosi, che prima svolgevano le attività a mano. Ben pochi di questi computer eseguono operazioni stimolanti: per lo più conservano informazioni banali, come registrazioni di fatture

o prenotazioni aeree. Semplicemente, ne contengono grandi quantità.

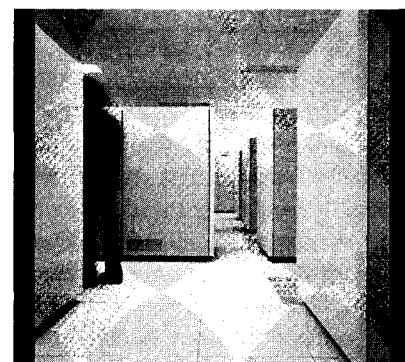
IBM non fu la prima società a costruire computer mainframe: questo primato appartiene a Univac Corporation. Tuttavia, ben presto IBM si guadagnò il ruolo principale, in parte grazie alla qualità tecnologica e all'attenzione posta alle necessità dei clienti, e in parte perché sfruttò il suo predominio, strutturando i suoi prodotti e i suoi servizi in modo da rendere difficile ai clienti intercambiare con quelli di altri fornitori. Negli anni '60, i concorrenti di IBM, i cosiddetti "Sette Nani" (GE, RCA, Univac, Honeywell, Burroughs, Control Data e NCR), se la videro brutta. Alcuni uscirono completamente dal mercato dei computer, mentre altri tentarono senza successo di combinare le loro forze. Le previsioni generali indicavano una sconfitta finale per tutti loro. Fu in questo clima che il governo degli Stati Uniti intentò una causa antitrust contro IBM, nel 1969. Il dibattimento iniziò nel 1975, trascinandosi fino al 1982, quando l'amministrazione Reagan l'abbandonò, dichiarandola "priva di valore".

Naturalmente, da allora il panorama informatico è cambiato completamente. Proprio come i dinosauri cedettero il passo a creature più piccole e agili, sono apparse tre nuove ondate di computer: i minicomputer, le workstation e i microcomputer, tutti progettati da nuove società e non dai Sette Nani. Al giorno d'oggi, il ruolo dei main-

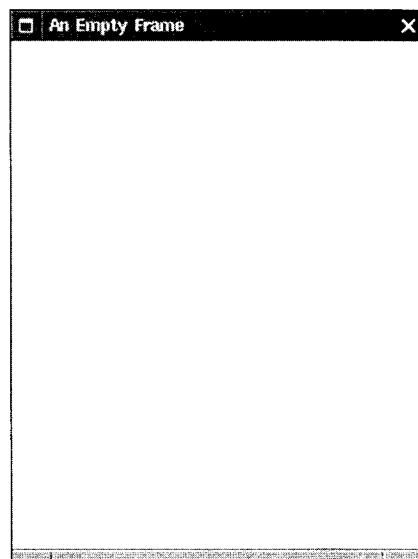
frame si è ridimensionato, e IBM, che è tuttora una grande società ricca di risorse, non domina più il mercato dei computer.

Oggi i mainframe sono ancora in uso per due ragioni. Innanzitutto, eccellono tuttora nell'elaborazione di grandi volumi di dati; secondariamente, ma in modo forse ancora più importante, i programmi per la gestione di dati commerciali si sono perfezionati nel corso di 30 e più anni, risolvendo, uno dopo l'altro, molti problemi. Trasferire questi programmi su computer meno costosi, con linguaggi e sistemi operativi diversi, è difficoltoso e genera errori. Negli anni '90 Sun Microsystems, un costruttore di primo piano nel settore workstation, era ansiosa di dimostrare che il sistema mainframe usato al proprio interno si poteva ridimensionare e sostituire con le proprie apparecchiature. Sun alla fine ci riuscì, ma impiegò oltre cinque anni, molto più di quanto avesse previsto.

#### Un computer mainframe



**Figura 22**  
Una finestra di tipo frame



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);
```

Questo frame sarà largo 300 pixel e alto 400 pixel (i pixel sono i minuscoli 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 indefinitivamente, 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 22.

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

Nei Capitoli 3, 9 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 una finestra di tipo frame.

### File ch02/emptyframe/EmptyFrameViewer.java

```
import javax.swing.JFrame;

public class EmptyFrameViewer
{
    public static void main(String[] args)
    {
        JFrame frame = new JFrame();

        frame.setSize(300, 400);
        frame.setTitle("An Empty Frame");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

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

### Auto-valutazione

27. Come si può visualizzare un frame quadrato con la scritta “Hello, World!” nella barra del titolo?
28. Come si possono visualizzare contemporaneamente due frame all’interno dello stesso programma?

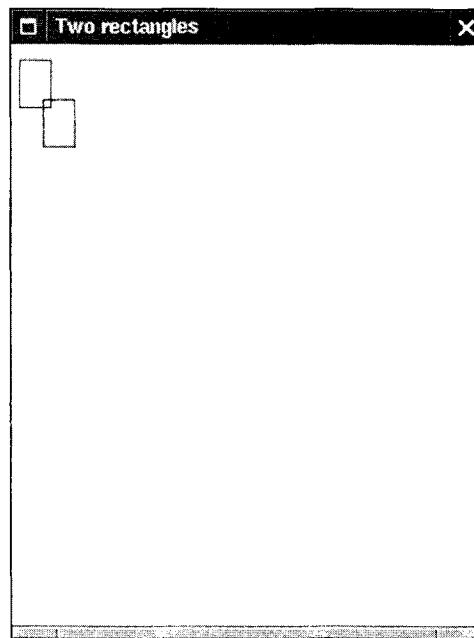
## 2.12 Disegnare all’interno di un componente



In questo paragrafo vedrete come tracciare figure all’interno di una finestra di tipo frame. La prima figura sarà davvero modesta, una coppia di rettangoli (Figura 23), ma presto imparerete a tracciare figure più interessanti. Lo scopo di questo esempio è quello di mostrarvi lo schema elementare di un programma che disegna figure: non è possibile disegnare direttamente in un frame e ogni volta che volete visualizzare qualcosa all’interno di esso, che sia un pulsante o una forma grafica, dovete costruire un oggetto di tipo *componente* e aggiungerlo al frame. 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 10.

**Figura 23**  
Disegnare rettangoli



Per il momento, seguite semplicemente lo schema qui proposto:

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

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 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 il tracciamento di testi, e così via.

La classe `Graphics` è, però, troppo scarna. 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 quei metodi più complessi che consentono di disegnare oggetti grafici bidimensionali.

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

Usate un cast per ottenere l'oggetto di tipo `Graphics2D` a partire dal parametro di tipo `Graphics` del metodo `paintComponent`.

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;
        ...
    }
}
```

I concetti di estensione di una classe e di operazione di cast saranno discussi nel Capitolo 10: per ora, inserite semplicemente il cast all'inizio dei vostri metodi `paintComponent`.

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);
        ...
    }
}
```

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 ch02/rectangles/RectangleComponent.java

```
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import javax.swing.JComponent;

/*
 * Un componente che disegna due rettangoli.
 */
public class RectangleComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        // recupera il riferimento a Graphics2D
        Graphics2D g2 = (Graphics2D) g;

        // costruisce un rettangolo e lo disegna
        Rectangle box = new Rectangle(5, 10, 20, 30);
        g2.draw(box);

        // costruisce un altro rettangolo e lo disegna
        Rectangle box2 = new Rectangle(30, 10, 20, 30);
        g2.draw(box2);
    }
}
```

```

        Rectangle box = new Rectangle(5, 10, 20, 30);
        g2.draw(box);

        // sposta il rettangolo di 15 unità verso destra e di 25 unità verso
        // il basso
        box.translate(15, 25);

        // disegna il rettangolo nella nuova posizione
        g2.draw(box);
    }
}

```

Per poter vedere i rettangoli disegnati sullo schermo, occorre ora visualizzare un frame contenente il componente, così come lo abbiamo appena progettato. Fate così:

1. Costruite un frame, come descritto nel paragrafo precedente.
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, come descritto nel paragrafo precedente.

Ecco il codice completo.

### File ch02/rectangles/RectangleViewer.java

```

import javax.swing.JFrame;

public class RectangleViewer
{
    public static void main(String[] args)
    {
        JFrame frame = new JFrame();

        frame.setSize(300, 400);
        frame.setTitle("Two rectangles");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        RectangleComponent component = new RectangleComponent();
        frame.add(component);

        frame.setVisible(true);
    }
}

```

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 stesso.



## Auto-valutazione

29. Come si può modificare il programma in modo che disegni due quadrati?
30. Come si può modificare il programma in modo che disegni un rettangolo e un quadrato?
31. Cosa succede se invocate `g.draw(box)` invece di `g2.draw(box)`?



## Argomenti avanzati 2.2

### Applet

**Si applet sono programmi eseguiti all'interno di un browser web.**

Nel paragrafo precedente avete visto come scrivere un programma che visualizza forme grafiche, ma alcune persone preferiscono usare gli *applet* per imparare la progettazione grafica. Gli applet hanno due vantaggi: non hanno bisogno di componenti a sé stanti né di classi di visualizzazione, ma possono essere realizzati con un'unica classe; inoltre, forse ancora più importante, gli applet possono essere eseguiti all'interno di un browser Web, consentendovi di collocare le vostre creazioni in una pagina Web visibile al mondo intero.

Per realizzare un applet dovete usare codice che segua questo schema:

```
public class MyApplet extends JApplet
{
    public void paint(Graphics g)
    {
        // recupera il riferimento a Graphics2D
        Graphics2D g2 = (Graphics2D) g;

        // istruzioni per disegnare
        ...
    }
}
```

Lo schema è molto simile a quello di un componente, con due differenze di poco conto:

1. Si deve estendere `JApplet` invece di `JComponent`.
2. Le istruzioni che tracciano il disegno devono essere inserite nel metodo `paint` invece che nel metodo `paintComponent`.

Questo applet disegna due rettangoli:

### File ch02/applet/RectangleApplet.java

```
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import javax.swing.JApplet;
```

```

/*
  Un applet che disegna due rettangoli.
*/
public class RectangleApplet extends JApplet
{
    public void paint(Graphics g)
    {
        // recupera il riferimento a Graphics2D
        Graphics2D g2 = (Graphics2D) g;

        // costruisce un rettangolo e lo disegna
        Rectangle box = new Rectangle(5, 10, 20, 30);
        g2.draw(box);

        // sposta il rettangolo di 15 unità verso destra e di 25 unità verso
        // il basso
        box.translate(15, 25);

        // disegna il rettangolo nella nuova posizione
        g2.draw(box);
    }
}

```

Per eseguire un applet occorre un file HTML che contenga un marcatore **applet**.

Per eseguire questo applet serve un file HTML che contenga un marcatore (*tag*) **applet**. Il linguaggio HTML (HyperText Markup Language) viene utilizzato per descrivere le pagine Web. Ecco, come esempio, il più semplice file HTML che è in grado di visualizzare il nostro applet che disegna rettangoli:

### File ch02/applet/RectangleApplet.html

```
<applet code="RectangleApplet.class" width="300" height="300">
</applet>
```

Se conoscete il linguaggio HTML, potete descrivere la vostra creazione con legittimo orgoglio, aggiungendo testo e altri marcatori HTML:

### File ch02/applet/RectangleAppletExplained.html

```
<html>
<head>
  <title>Two rectangles</title>
</head>
<body>
  <p>Here is my <i>first applet</i>:</p>
  <applet code="RectangleApplet.class" width="300" height="300">
  </applet>
</body>
</html>
```

Un file HTML può anche contenere più applet: basta aggiungere un diverso marcatore **applet** per ogni applet.

Potete assegnare al file HTML il nome che preferite, ma si può semplicemente usare lo stesso nome dell'applet. Tuttavia, alcuni ambienti di sviluppo generano già un file

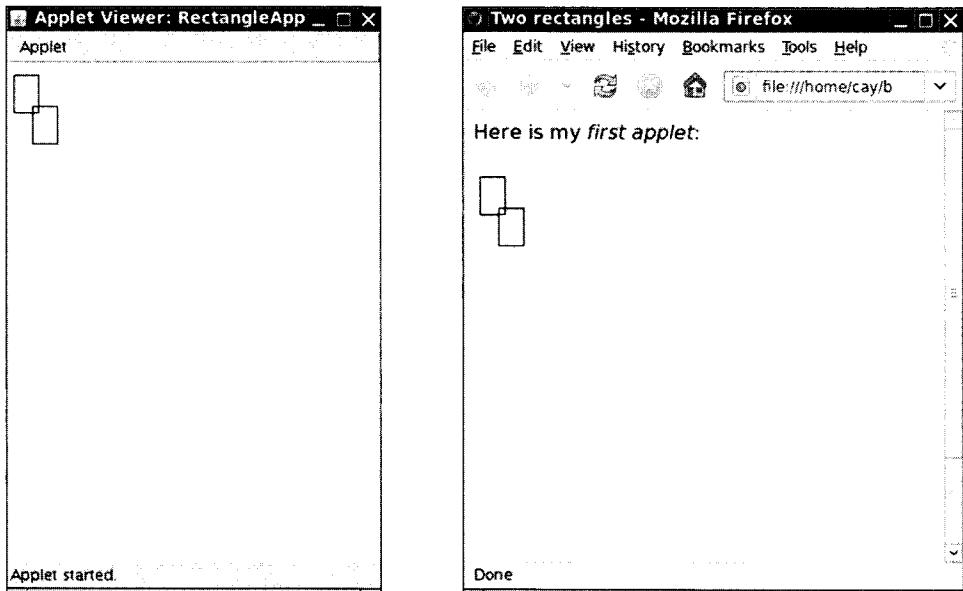
HTML con lo stesso nome del progetto, per contenere le note del lavoro: in questo caso, dovete assegnare un nome diverso al file HTML che contiene il vostro applet.

Per eseguire l'applet, esistono due possibilità. Potete usare il *visualizzatore di applet*, un programma contenuto nel Java Software Development Kit distribuito da Sun Microsystems. È sufficiente avviare il visualizzatore e fornirgli il nome del file HTML che contiene i vostri applet, in questo modo:

```
appletviewer RectangleApplet.html
```

Il visualizzatore di applet mostra solamente l'applet, ignorando tutti gli altri tag HTML.

Un applet  
nel visualizzatore di applet  
(a sinistra)  
  
Un applet in un browser  
Web  
(a destra)

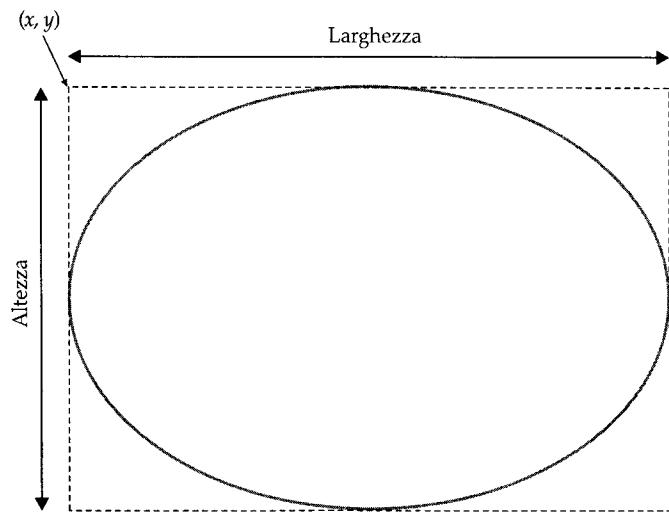


Potete anche visualizzare l'applet all'interno di qualsiasi browser Web abilitato all'uso di Java, come Firefox o Safari (se usate Internet Explorer, avrete probabilmente bisogno di configurarlo in modo opportuno, perché la configurazione standard di Microsoft contiene una versione di Java non aggiornata, oppure addirittura non contiene Java; seguite le istruzioni presenti nel sito [java.com](http://java.com)). La figura di destra mostra l'applet visualizzato in un browser: come potete vedere, sono presenti sia il testo, sia l'applet.

## 2.13 Ellissi, linee, testo e colore

Nel Paragrafo 2.12 avete imparato a scrivere un programma che disegna rettangoli: qui imparerete a tracciare altre forme geometriche, come ellissi e linee. Grazie a questi elementi grafici, potrete comporre immagini piuttosto interessanti.

**Figura 24**  
Un'ellisse  
e il suo rettangolo  
di delimitazione



### 2.13.1 Ellissi e cerchi

Per disegnare un'ellisse, dovete specificare il suo *rettangolo di delimitazione* (osservate la Figura 24), analogamente a come definireste un rettangolo, cioè tramite le coordinate *x* e *y* dell'angolo superiore sinistro, la larghezza (*width*) e l'altezza (*height*).

Tuttavia, non potete usare una semplice classe `Ellipse`, che non esiste: dovete, 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);
```

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

Il nome della classe, `Ellipse2D.Double`, appare diverso dai nomi di classe che avete incontrato 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* di `Ellipse2D`. In realtà, quando costruirete e manipolate ellissi non dovete preoccuparvi 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 all'angolo superiore sinistro del rettangolo di delimitazione, *non* il centro del cerchio.

### 2.13.2 Linee

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.

### 2.13.3 Scrivere testo

**Il metodo `drawString` disegna una stringa, iniziando 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`. Dovete specificare come parametri la stringa e le coordinate  $x$  e  $y$  del punto base del primo carattere della stringa stessa (osservate la Figura 25), come in questo esempio:

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

**Figura 25**

Il punto base e la linea base



## 2.13.4 Colori

**Quando impostate un nuovo colore nel contesto grafico, esso viene usato nelle operazioni grafiche successive.**

Quando iniziate a disegnare per la prima volta, 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 (Red, Green, Blue)*: questo significa che per specificare un colore si indica la quantità dei *colori primari* (rosso, verde e blu) che generano il colore desiderato. Le quantità sono espresse da numeri interi compresi tra 0 (colore primario non presente) e 255 (quantità massima). Per esempio, l'enunciato seguente costruisce un oggetto di tipo `Color` mediante una quantità massima di rosso e di blu e 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 un rettangolo di colore diverso dal nero, dovete per prima cosa impostare il colore nell'oggetto di tipo `Graphics2D`, poi invocarne il metodo `draw`:

```
g2.setColor(Color.RED); // imposta il colore rosso
g2.draw(box); // 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 rettangolo con il colore attivo nel contesto grafico:

```
g2.fill(box);
```

**Tabella 4**  
Colori predefiniti

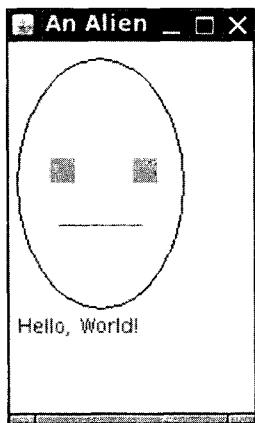
Colore	Descrizione	Valore RGB
<code>Color.BLACK</code>	Nero	0, 0, 0
<code>Color.BLUE</code>	Blu	0, 0, 255
<code>Color.CYAN</code>	Azzurro ciano	0, 255, 255
<code>Color.GRAY</code>	Grigio	128, 128, 128
<code>Color.DARK_GRAY</code>	Grigio scuro	64, 64, 64
<code>Color.LIGHT_GRAY</code>	Grigio chiaro	192,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

Il seguente programma utilizza tutte le forme viste, creando il semplice disegno che potete vedere nella Figura 26.

### File ch02/face/FaceComponent.java

```
import java.awt.Color;
import java.awt.Graphics;
```

**Figura 26**  
visto di un alieno



```
import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Line2D;
import javax.swing.JComponent;

/*
   Un componente che disegna il viso di un alieno.
*/
public class FaceComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        // recupera il riferimento a Graphics2D
        Graphics2D g2 = (Graphics2D) g;

        // disegna la testa
        Ellipse2D.Double head = new Ellipse2D.Double(5, 10, 100, 150);
        g2.draw(head);

        // disegna gli occhi
        g2.setColor(Color.GREEN);
        Rectangle eye = new Rectangle(25, 70, 15, 15);
        g2.fill(eye);
        eye.translate(50, 0);
        g2.fill(eye);

        // disegna la bocca
        Line2D.Double mouth = new Line2D.Double(30, 110, 80, 110);
        g2.setColor(Color.RED);
        g2.draw(mouth);

        // scrive il saluto
        g2.setColor(Color.BLUE);
        g2.drawString("Hello, World!", 5, 175);
    }
}
```



## Note di cronaca 2.2

### L'evoluzione di 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 lontani: 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 Internet: un insieme di reti interconnesse. Tutte le reti appartenenti a Internet condividono protocolli

comuni per la trasmissione dei dati: Kahn e Vinton Cerf svilupparono questo insieme di protocolli, oggi denominati 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 GNU ("Gnu's Not Unix") produce un insieme di programmi di utilità per sistemi operativi e di strumenti di sviluppo di ottima qualità ([www.gnu.org](http://www.gnu.org)), mentre 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 iper-testuali, che consentono agli utenti una consultazione arricchita

da collegamenti con documenti correlati: l'infrastruttura derivante è 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 fondè, 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.

Il browser Mosaic  
di NCSA

The screenshot shows the NCSA Mosaic web browser window. The title bar reads "NCSA Mosaic - Virginia Tech Chemistry Hypermedia". The menu bar includes File, Edit, Options, Navigate, Annotate, Chemistry, News/http, Starting Pts, and Help. Below the menu is a toolbar with icons for Back, Forward, Stop, Home, and others. The main content area displays a page titled "Virginia Tech Chemical Education Hypermedia". The page text states: "The main headings below lead to descriptions of the hypermedia tutorials that are listed in the menus. A separate Virginia Tech Chemistry Course Material document indexes course-specific material for Va Tech chemistry students." Below this is a section titled "Overview of the Chemistry Hypermedia Project" with a link. Further down is a section titled "Analytical Chemistry Hypermedia" with links to "Introduction to Analytical Chemistry", "Analytical Chemistry Basics", "Encyclopedia of Instrumental Methods", and "Analytical Spectroscopy". At the bottom is another section titled "Organic Chemistry Hypermedia".

### File ch02/face/FaceViewer.java

```
import javax.swing.JFrame;

public class FaceViewer
{
    public static void main(String[] args)
    {
        JFrame frame = new JFrame();
        frame.setSize(150, 250);
        frame.setTitle("An Alien Face");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        FaceComponent component = new FaceComponent();
        frame.add(component);

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

### Auto-valutazione

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

## Riepilogo degli obiettivi di apprendimento

### Utilizzo di numeri interi e in virgola mobile

- Un tipo specifica un insieme di valori e le operazioni che possono essere compiute su di essi.
- Il tipo `double` caratterizza numeri in virgola mobile, che possono avere una parte frazionale.
- In Java, i tipi numerici sono tipi primitivi e i numeri non sono oggetti.
- Si possono combinare numeri in espressioni usando operatori aritmetici, quali +, - e \*.

### Dichiarazione di variabili in Java

- Le variabili vengono usate per memorizzare valori che si vogliono utilizzare in seguito. Ogni variabile è caratterizzata da un tipo, un nome e un contenuto.
- Gli identificatori di variabili, metodi e classi sono composti di lettere, cifre e segni di sottolineatura.
- Per convenzione, i nomi delle variabili dovrebbero iniziare con una lettera minuscola.

### Procedura di assegnazione di valori a variabili e loro inizializzazione

- Per modificare il valore di una variabile si usa l'operatore di assegnazione (=).
- Tutte le variabili devono essere inizializzate prima di essere utilizzate.

### Dichiarazione di 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 accede ai dati di un oggetto.
- Una classe dichiara i metodi che possono essere applicati ai suoi oggetti.
- 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.

### Valori restituiti dai metodi e loro parametri impliciti ed esplicativi

- Un parametro è un dato fornito in ingresso a un metodo.
- Il parametro隐式的 dell'invocazione di un metodo è l'oggetto con cui viene invocato il metodo stesso. Tutti gli altri parametri si dicono esplicativi.
- Il valore restituito da un metodo è il risultato da esso calcolato, da utilizzare nel codice che ha invocato il 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 di tipo frame

- Per visualizzare un frame si costruisce un oggetto di tipo `JFrame`, se ne impostano le dimensioni e lo si rende visibile.
- Per visualizzare qualunque cosa 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.
- Usate un cast per ottenere l'oggetto di tipo `Graphics2D` a partire dal parametro di tipo `Graphics` del metodo `paintComponent`.

- Gli applet sono programmi che vengono eseguiti all'interno di un browser Web.
- Per eseguire un applet occorre un file HTML che contenga un marcatore `applet`.
- Gli applet possono essere visualizzati con un apposito visualizzatore o con un browser abilitato al linguaggio Java.

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

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

## Classi, oggetti e metodi presentati nel 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 ripasso

- ★ **Esercizio R2.1.** Spiegate la differenza fra un oggetto e un riferimento a oggetto.
- ★ **Esercizio R2.2.** Spiegate la differenza fra un oggetto e una variabile oggetto.
- ★ **Esercizio R2.3.** Spiegate la differenza fra un oggetto e una classe.
- ★★ **Esercizio R2.4.** Fornite il codice Java per costruire un *oggetto* della classe `Rectangle` e per dichiarare una *variabile oggetto* della stessa classe.
- ★★ **Esercizio R2.5.** Illustrate il significato del simbolo `=` in Java e in matematica.
- ★★ **Esercizio R2.6.** Fornite il codice Java per costruire i seguenti oggetti:
  - 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.
- ★★ **Esercizio R2.7.** Ripetete l'Esercizio precedente, definendo ora variabili oggetto che vengano inizializzate con gli oggetti appena costruiti.

- \*\* **Esercizio R2.8.** Scrivete un enunciato Java che inizializzi una variabile `square` con un rettangolo il cui angolo superiore sinistro abbia coordinate (10, 20) e i cui lati abbiano lunghezza 40. Scrivete, poi, un enunciato che sostituisca il contenuto di `square` con un rettangolo avente le stesse dimensioni ma angolo superiore sinistro posizionato nel punto (20, 20).
- \*\* **Esercizio R2.9.** Scrivete enunciati Java che inizializzino due variabili `square1` e `square2` in modo che facciano riferimento al medesimo quadrato, i cui lati abbiano lunghezza 40 e il cui centro sia posizionato nel punto di coordinate (20, 20).
- \*\* **Esercizio R2.10.** Scrivete enunciati Java che inizializzino la variabile stringa `message` con "Hello". modificandone poi il contenuto in "HELLO". Usate il metodo `toUpperCase`.
- \*\* **Esercizio R2.11.** Scrivete enunciati Java che inizializzino la variabile stringa `message` con "Hello". modificandone poi il contenuto in "hello". Usate il metodo `replace`.
- \*\* **Esercizio R2.12.** Identificate gli errori presenti negli enunciati seguenti:
  - a. `Rectangle r = (5, 10, 15, 20);`
  - b. `double width = Rectangle(5, 10, 15, 20).getWidth();`
  - c. `Rectangle r;`  
`r.translate(15, 25);`
  - d. `r = new Rectangle();`  
`r.translate("far, far away!");`
- \* **Esercizio R2.13.** Indicate due metodi d'accesso e due metodi modificatori della classe `Rectangle`.
- \*\* **Esercizio R2.14.** Consultate la documentazione API della classe `Rectangle` e identificate il metodo

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

Leggete con attenzione la documentazione, poi stabilite quale sia il risultato dell'esecuzione degli enunciati seguenti:

```
Rectangle box = new Rectangle(5, 10, 20, 30);
box.add(0, 0);
```

Se non siete sicuri, scrivete un piccolo programma di collaudo.

- \*G **Esercizio R2.15.** Spiegate la differenza fra un'applicazione grafica e un'applicazione per console.
- \*\*G **Esercizio R2.16.** Chi invoca il metodo `paintComponent` di un componente grafico? Quando si verifica la chiamata di tale metodo?
- \*\*G **Esercizio R2.17.** Perché il parametro del metodo `paintComponent` è di tipo `Graphics`, anziché `Graphics2D`?
- \*\*G **Esercizio R2.18.** A cosa serve un contesto grafico?
- \*\*G **Esercizio R2.19.** Perché nei programmi grafici usiamo classi separate per i componenti e per la loro visualizzazione?
- \*G **Esercizio R2.20.** In che modo si specifica il colore del testo?

# 3

## Realizzare classi

### Obiettivi del capitolo

- Acquisire familiarità con il procedimento di realizzazione di classi
- Essere in grado di progettare 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 possano 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 verificare che la vostra classe funzioni correttamente.

### 3.1 Variabili di esemplare

Nel Capitolo 2 avete visto come si usano oggetti di classi esistenti, ora inizierete a realizzare vostre classi. Iniziamo con 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. Apprenderete, poi, un procedimento sistematico per la realizzazione di classi.

Il nostro primo esempio riguarda una classe che rappresenta un *contapersone* (“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).

Ogni volta che l’operatore preme un pulsante, il valore del conteggio viene incrementato di un’unità: modelliamo questa operazione mediante il metodo `count`. Un contapersone reale visualizza il conteggio attuale: nella nostra simulazione, useremo, invece, il metodo `getValue` per ispezionare il valore di conteggio.

```
Counter tally = new Counter();
tally.count();
tally.count();
int result = tally.getValue(); // imposta result al valore 2
```

Per realizzare la classe `Counter`, dobbiamo decidere quali dati vengano memorizzati all’interno di un oggetto contatore. In questo esempio, così semplice, tale indagine è davvero elementare: ogni contatore ha la necessità di usare una variabile per tener traccia del numero di avanzamenti del conteggio che sono stati richiesti mediante il pulsante.

**Figura 1**  
Un contapersone



Un oggetto usa variabili di esemplare per memorizzare i dati necessari per l’esecuzione dei propri metodi.

Un oggetto memorizza i propri dati all’interno di *variabili di esemplare* (o di *istanza*). Un *esemplare* di una classe è quello che, fino ad ora, abbiamo chiamato “un oggetto della classe”. Una variabile di esemplare è, quindi, una zona di memorizzazione presente in ogni oggetto della classe.

La dichiarazione della classe specifica le sue variabili di esemplare:

```
public class Counter
{
    private int value;
    ...
}
```

La dichiarazione di una variabile di esemplare è così composta:

- Una modalità d’accesso (`private`)

# Sintassi di Java

## 3.1 Dichiarazione di variabile di esemplare

### Sintassi

```
modalitàDiAccesso class NomeClasse
{
    modalitàDiAccesso 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.

- 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.

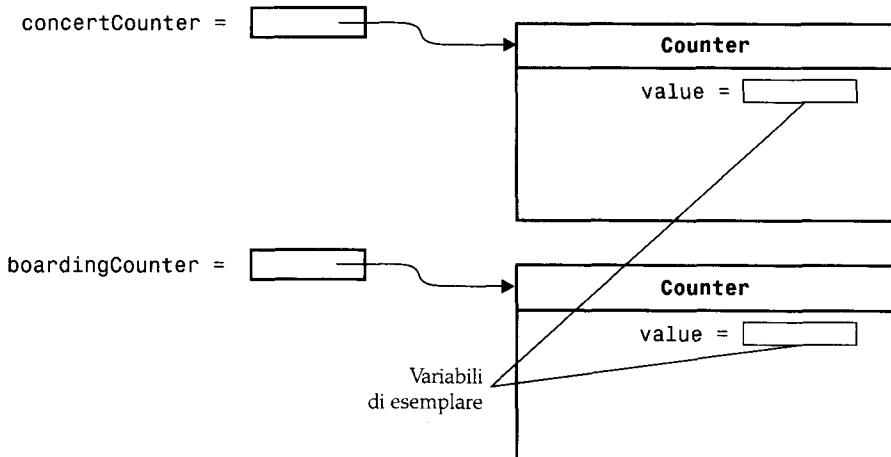
Ciascun oggetto di una classe ha il proprio insieme di variabili di esemplare. Ad esempio, se `concertCounter` e `boardingCounter` sono due oggetti della classe `Counter`, ognuno di essi ha la propria variabile `value`, come rappresentato in Figura 2. Come vedrete nel Paragrafo 3.7, nel momento in cui viene costruito un oggetto di tipo `Counter` la sua variabile di esemplare `value` viene impostata al valore 0.

Per capire meglio come i metodi interagiscano con le variabili di esemplare, diamo uno sguardo a come verranno realizzati i metodi della classe `Counter`. Il metodo `count` incrementa di un'unità il valore del contatore. Nel Paragrafo 3.3 parleremo della sintassi relativa all'intestazione del metodo, mentre qui ci concentriamo sul suo corpo, all'interno delle parentesi graffe:

```
public void count()
{
```

**Figura 2**

variabili di esemplare



```
    value = value + 1;
}
```

Osservate come il metodo `count` 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.count();
```

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` è uno speciale enunciato che pone termine all'invocazione del metodo e restituisce un risultato a chi ha invocato il metodo stesso.

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 `balance` si può accedere dai metodi `count` 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`.

Nel prossimo paragrafo motiveremo la scelta di rendere private le variabili di esemplare.

**Alle variabili di esemplare private possono accedere soltanto i metodi appartenenti alla medesima classe.**



### Auto-valutazione

1. Scrivete il corpo di un metodo, `public void reset()`, che faccia tornare a zero il valore di conteggio di un contatore.
2. 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?

## 3.2 Incapsulamento

Nel paragrafo precedente avete imparato a nascondere le variabili di esemplare, rendendole private. Ora, la domanda è: perché mai un programmatore dovrebbe voler nascondere qualcosa? In questo paragrafo discuteremo proprio i vantaggi derivanti dalla pratica di tenere nascoste le informazioni.

Nascondere le informazioni (“information hiding”) non è una strategia adottata soltanto nella programmazione di calcolatori: viene, invece, usata in molte discipline dell’ingegneria. Considerate la centralina elettronica presente in tutte le moderne automobili: si tratta di un dispositivo che controlla gli istanti di accensione delle candele e il flusso di carburante verso il motore. Se chiedete al vostro meccanico di fiducia che cosa ci sia all’interno del modulo elettronico, lo vedrete dubbioso.

Tale modulo è, per lui, una *scatola nera* (“black box”), qualcosa che svolge quasi magicamente i propri compiti. Un meccanico 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: è ben definita la sua interfaccia con il mondo esterno. Ad esempio, il meccanico dell’automobile sa come il modulo di controllo elettronico deve essere collegato ai sensori e agli altri dispositivi che costituiscono il motore.

Il processo che nasconde i dettagli realizzativi, rendendo invece pubblica un’interfaccia, si chiama *incapsulamento*. 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 i costruttori di automobili mettono scatole nere all’interno dei loro prodotti? Le scatole nere semplificano molto il lavoro dei meccanici: prima che venissero inventate le centraline elettroniche, il flusso di carburante veniva regolato da un dispositivo meccanico, il carburatore, la cui messa a punto e riparazione era notoriamente molto complessa. Oggi, un meccanico non ha più bisogno di sapere cosa ci sia all’interno della centralina.

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, quindi, migliaia di metodi, ma se si identifica 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 usando per comporre programmi: trattandoli, cioè, da scatole nere. Il vostro ruolo, quindi, è stato sostanzialmente analogo a quello di un meccanico che aggiusta un’automobile sostituendone la centralina elettronica.

In questo capitolo passeremo alla progettazione di classi. In questi paragrafi, il vostro ruolo è analogo a quello del progettista di componenti per automobili, che progetta la centralina elettronica a partire da transistori, condensatori e altri componenti elettronici. Apprenderete le tecniche di programmazione Java necessarie per consentire ai vostri oggetti di realizzare il comportamento desiderato.

## Auto-valutazione

- Considerate la classe `Counter`. Il valore di conteggio parte da zero e viene incrementato dal metodo `count`, 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 dati 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?



Fonte: Creative Commons Attribution-NonCommercial-ShareAlike license

L’incapsulamento prevede di  
nascondere i dettagli realizzativi,  
metodi per l’accesso ai dati.

L’incapsulamento consente  
programmatori di usare una classe  
doverne conoscere i dettagli  
realizzativi.

L’incapsulamento agevola  
la localizzazione di errori  
modifica di dettagli realizzativi  
di una classe.

### 3.3 Progettare l'interfaccia pubblica di una classe

Per realizzare una classe, occorre prima individuare quali metodi siano necessari.

In questo paragrafo analizzeremo il processo 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.

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 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 semplici invocazioni, 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*, composto dagli enunciati che vengono eseguiti quando il metodo viene invocato.

```
public void deposit(double amount)
{
    corpo, che verrà riempito in seguito
}
```

Nel Paragrafo 3.5 scriveremo il corpo dei metodi.

Ogni intestazione di metodo contiene le seguenti componenti:

- Una *modalità di accesso* (solitamente `public`)
- Il *tipo restituito* (cioè il tipo del valore restituito, come `void` o `double`)
- Il nome del metodo (come `deposit`)
- Racchiuso fra parentesi tonde, un elenco delle *variabili parametro* del metodo (se ce ne sono), come `double amount`

La modalità di accesso determina quali altri metodi possono invocare questo metodo. La maggior parte dei metodi dovrebbe essere dichiarata `public`: in questo modo, tutti gli altri metodi nei vostri programmi possono invocarli (talvolta può essere utile avere metodi `private`, che possono essere invocati soltanto da altri metodi della stessa classe).

Il tipo restituito indica il tipo di dato del valore che il metodo restituisce. Il metodo `deposit` non restituisce alcun valore, mentre il metodo `getBalance` restituisce un valore di tipo `double`.

Ogni parametro del metodo è caratterizzato da un tipo e da un nome, che ne descrive il significato. Ad esempio, il metodo `deposit` ha un unico parametro di nome `amount` e di tipo `double`.

Dovete poi fornire *costruttori*. Un costruttore fornisce un valore iniziale alle variabili di esemplare di un oggetto e, in Java, 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 hanno un tipo restituito (nemmeno `void`)

Vogliamo 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);
```

Esattamente come un metodo, anche un costruttore ha un corpo, cioè una sequenza di enunciati che viene eseguita quando viene costruito un nuovo oggetto.

```
public BankAccount()
{
    corpo, 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 3.5.

~~L'intestazione di un metodo ne indica il tipo restituito, il nome e i parametri (loro tipo e nome).~~

~~Costruttori impostano i valori iniziali per i dati degli oggetti. Il nome di un costruttore è sempre uguale al nome della classe.~~

Non preoccupatevi del fatto che esistano due costruttori con lo stesso nome: *tutti i* costruttori di una classe hanno lo stesso nome, che è il nome della classe. Il compilatore è in grado di distinguerli, perché richiedono parametri diversi.

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

    // 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*: sono le operazioni che qualsiasi programmatore può utilizzare per creare e manipolare oggetti di tipo `BankAccount`.

La nostra classe `BankAccount` è semplice, ma consente ai programmatori 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 da un conto a un altro
double transferAmount = 500;
momsSavings.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 programmatore 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 3.5, ma, prima di quello, dobbiamo esaminare ancora un elemento: la *documentazione* dell'interfaccia pubblica, che sarà argomento del prossimo paragrafo.

## 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?

## 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.

## 3.4 Commentare l'interfaccia pubblica

Nel realizzare classi e metodi, dovreste prendere l'abitudine di *commentare* esaurientemente il loro comportamento. In Java, per i *commenti di documentazione* esiste un formato standard, molto utile. Se nelle vostre classi usate tale formato, potrete usare il programma `javadoc` per generare automaticamente un insieme chiaro ed elegante di pagine HTML che le descrivono (si vedano i Consigli per la produttività 3.1 per una descrizione di questo programma di utilità).

**Usate i commenti di documentazione per descrivere le classi e i metodi pubblici dei vostri programmi.**

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 viene usato dal programma di utilità javadoc. Di seguito descrivete, per prima cosa, lo *scopo* del metodo. Poi, per ciascun parametro del metodo, inserite una riga che inizia con il marcitore `@param`, seguito dal nome del parametro e da una sua breve spiegazione. Infine, inserite 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 di utilità javadoc copia la *prima* frase di ciascun commento in una tabella riassuntiva all'interno della documentazione HTML, per cui è bene scrivere tale frase 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 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)
{
    realizzazione (completata in seguito)
}

/**
 * Ispeziona il saldo attuale del conto corrente.
 * @return il saldo attuale
 */
public double getBalance()
{
    realizzazione (completata in seguito)
}
```

I commenti che avete appena visto descrivono singoli *metodi*. Fornite anche un breve commento per ciascuna *classe*, per spiegarne lo scopo. La sintassi per il commento di documentazione di una classe è molto semplice: basta inserirlo all'inizio della classe.

```
/**
 * Un conto bancario ha un saldo che può essere modificato
 * da depositi 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.

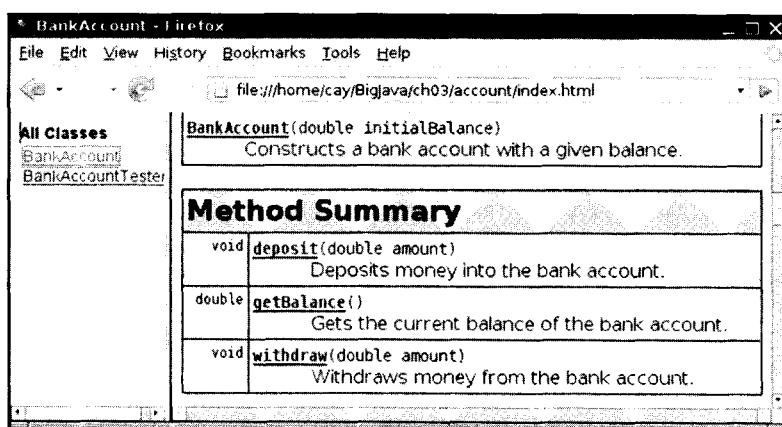
**Scrivete commenti di documentazione per ogni classe, ogni metodo, ogni parametro e ogni valore restituito.**

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 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 (in Figura 3), mentre i commenti di tipo `@param` e `@return` vanno a comporre la descrizione dettagliata di ciascun metodo (in Figura 4). Se qualcuno di tali commenti viene omesso, il programma `javadoc` genera documenti con strani spazi vuoti.

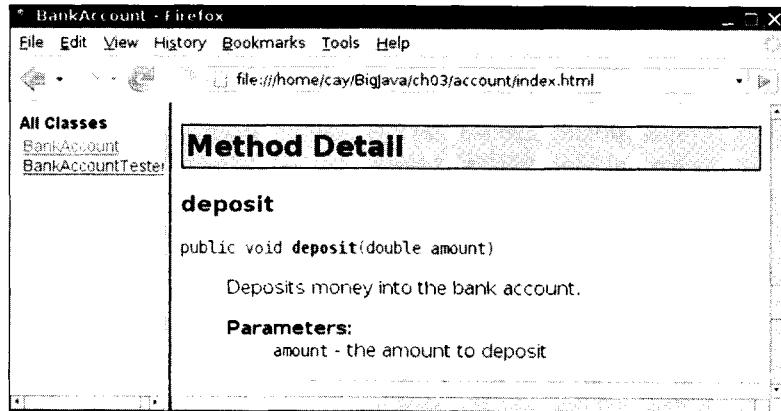
**Figura 3**

Assunto dei metodi generato da javadoc



**Figura 4**

La documentazione del metodo generata da javadoc



Questo formato della documentazione vi dovrebbe risultare familiare: i programmati che realizzano la libreria 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.



## Auto-valutazione

9. Aggiungete i commenti di documentazione alla classe `Counter` vista nel Paragrafo 3.1.
10. Supponete di aver migliorato la classe `BankAccount` in modo che a ciascun conto sia associato un numero di conto. Scrivete un commento per la documentazione del seguente costruttore:

```
public BankAccount(int accountNumber, double initialBalance)
```

11. 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()
```



## Consigli per la produttività 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 incorporare marcatori HTML nei commenti, per specificare font o per aggiungere immagini. Cosa forse ancora più importante, `javadoc` fornisce automaticamente *collegamenti ipertestuali* ad altre classi e metodi.

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 aggiornerete 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.

## 3.5 Realizzare la classe

Ora che abbiamo ben compreso le specifiche dell'interfaccia pubblica della classe `BankAccount`, passiamo alla sua realizzazione.

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 attuale (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;
    ...
}
```

Dopo aver individuato le variabili di esemplare, completiamo la classe `BankAccount` scrivendo i corpi di costruttori e metodi. Ogni corpo è composto da una sequenza di enunciati. Inizieremo con i costruttori, perché sono veramente banali, dal momento che un costruttore ha un compito assai 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:

```
BankAccount harrysChecking = new BankAccount(1000);
```

L'implementazione privata  
una classe comprende le variabili  
esemplare e il corpo di costruttori  
e metodi.

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

Le singole azioni elementari che avvengono durante l'esecuzione dell'enunciato sono queste:

- Creazione di un nuovo oggetto di tipo `BankAccount`.
- Invocazione del secondo costruttore (perché è stato fornito un parametro di costruzione).
- Assegnazione del valore 1000 alla variabile parametro `initialBalance`.
- Assegnazione del valore di `initialBalance` alla variabile di esemplare `balance` dell'oggetto appena creato.
- Restituzione, come valore dell'espressione `new`, di un riferimento a un oggetto, che è la posizione in memoria dell'oggetto appena creato.
- Memorizzazione nella variabile `harrysChecking` del riferimento all'oggetto.

Passiamo ora alla realizzazione dei metodi di `BankAccount`, iniziando dal metodo `deposit`:

```
public void deposit(double amount)
{
    balance = balance + amount;
}
```

Per capire esattamente come funziona il metodo, esaminiamo questo enunciato:

```
harrysChecking.deposit(500);
```

la cui esecuzione richiede le seguenti azioni:

- Assegnazione del valore 500 alla variabile parametro `amount`.
- Lettura della variabile di esemplare `balance` appartenente all'oggetto che si trova nella posizione memorizzata nella variabile `harrysChecking`.
- Addizione tra il valore di `amount` e il valore di `balance`.
- Memorizzazione della somma nella variabile di esemplare `balance`, sovrascrivendo il vecchio valore.

Il metodo `withdraw` è molto simile al metodo `deposit`:

```
public void withdraw(double amount)
{
    balance = balance - amount;
}
```

Ci manca un solo altro 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;
}
```

## 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; } Realizzazione privata.

    public void count() { value = value + 1; }
    public int getValue() { return value; }
}
```

**Interfaccia pubblica.**

## Sintassi di Java 3.3 Dichiarazione di metodo

### Sintassi

```
modalitàDiAccesso tipoRestituito nomeMetodo(tipoParametro nomeParametro, ...)
{
    corpo del metodo
}
```

### Esempio

Ci sono questi metodi fanno parte dell'interfaccia pubblica.

Questo metodo non restituisce alcun valore.

```
public void deposit(double amount)
{
    balance = balance + amount;
}
```

Un metodo modificatore modifica una variabile di esemplare.

```
public double getBalance()
{
    return balance;
}
```

Questo metodo non ha parametri.

Un metodo d'accesso restituisce un valore.

Abbiamo così completato la realizzazione 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.

**File ch03/account/BankAccount.java**

```
/**  
 * Un conto bancario ha un saldo che può essere modificato  
 * da depositi e prelievi.  
 */  
public class BankAccount  
{  
    private double balance;  
  
    /**  
     * Costruisce un conto bancario con saldo uguale a zero.  
     */  
    public BankAccount()  
{  
        balance = 0;  
    }  
  
    /**  
     * Costruisce un conto bancario con saldo assegnato.  
     * @param initialBalance il saldo iniziale  
     */  
    public BankAccount(double initialBalance)  
{  
        balance = initialBalance;  
    }  
  
    /**  
     * Versa denaro nel conto bancario.  
     * @param amount l'importo da versare  
     */  
    public void deposit(double amount)  
{  
        balance = balance + amount;  
    }  
  
    /**  
     * Preleva denaro dal conto bancario.  
     * @param amount l'importo da prelevare  
     */  
    public void withdraw(double amount)  
{  
        balance = balance - amount;  
    }  
  
    /**  
     * Ispeziona il valore del saldo attuale del conto bancario.  
     * @return il saldo attuale  
     */  
    public double getBalance()  
{  
        return balance;  
    }  
}
```

## Auto-valutazione

12. 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?
13. Perché con l'esecuzione di questa classe non riuscite a rubare soldi al conto bancario di vostra madre?

```
public class BankRobber
{
    public static void main(String[] args)
    {
        BankAccount momssSavings = new BankAccount(1000);
        momssSavings.balance = 0;
    }
}
```

14. La classe `Rectangle` ha quattro variabili di esemplare: `x`, `y`, `width` e `height`. Scrivete una possibile realizzazione del metodo `getWidth`.
15. Scrivete una possibile realizzazione del metodo `translate` della classe `Rectangle`.

## Consigli pratici 3.1

### Realizzare una classe

Questa sezione speciale di Consigli pratici (nell'originale, “How To”) vi dirà come realizzare una classe a partire da specifiche assegnate.

Ad esempio, in un compito a casa vi si potrebbe chiedere di progettare una classe che costituisca un modello di registratore di cassa. La classe dovrebbe consentire a un cassiere di digitare i prezzi di articoli e la quantità di denaro pagata dal cliente, calcolando il resto dovuto.

#### Fase 1 Identificare i metodi che vi viene chiesto di mettere a disposizione

Nell'esempio del registratore di cassa, non dovrete realizzare tutte le caratteristiche di un vero registratore di cassa, sono troppe: il compito assegnato vi specifica, in linguaggio naturale, *quali aspetti* di un registratore di cassa devono essere simulati dalla vostra classe. Fatene un elenco.

- Registra il prezzo di vendita per un articolo acquistato.
- Registra la somma di denaro pagata.
- Calcola il resto dovuto al cliente.

#### Fase 2 Specificare l'interfaccia pubblica

Trasformate l'elenco della 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.enterPayment(50);
double change = register.giveChange();
```

A questo punto abbiamo l'elenco dei metodi.

- `public void recordPurchase(double amount)`
- `public void enterPayment(double amount)`
- `public double giveChange()`

Per completare l'interfaccia pubblica dovete ancora specificare i costruttori. Chiedetevi quali informazioni vi siano necessarie per costruire un oggetto della vostra classe. A volte vorrete avere due costruttori: uno che imposti tutte le variabili di esemplare a valori predefiniti e un altro che usi a tale scopo valori definiti dall'utente.

Nel caso dell'esempio 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 per l'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)
    {
    }

    /**
     * Registra la quantità di denaro ricevuta come pagamento.
     * @param amount l'ammontare del pagamento
     */
    public void enterPayment(double amount)
```

```

    {
}

/**
    Calcola il resto dovuto al cliente e azzera la macchina
    in attesa del cliente 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. Ricordate 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 avete bisogno per portare a termine il compito assegnato al metodo. Create variabili di esemplare per conservare le informazioni necessarie ai metodi.

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 vostra classe, uno alla volta, iniziando dai più semplici. Ecco, ad esempio, la realizzazione del metodo `recordPurchase`:

```

public void recordPurchase(double amount)
{
    purchase = purchase + amount;
}

```

Ecco, invece, 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.

Dopo aver terminato la realizzazione, compilate la vostra classe e correggete tutti gli errori di compilazione.

#### 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 identificato 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.enterPayment(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
```

In alternativa, se usate un programma che vi consente di collaudare oggetti interattivamente, come BlueJ, costruite un oggetto e applicatevi le invocazioni dei metodi.



## Esempi completi 3.1

### Realizzare un semplice menu

Avete il compito di progettare la classe `Menu`. Un oggetto di tale classe è 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, mano a mano che queste vengono aggiunte al menu.

**Fase 1** Identificare i metodi che vi viene chiesto di mettere a disposizione

La descrizione del problema elenca due diversi compiti.

- Visualizza il menu.
- Aggiunge 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 in Consigli pratici 3.1, iniziamo scrivendo 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 la prima opzione al menu, dato che, in fin dei conti, non ha alcun senso disporre di 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 per l'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 del menu.
     @param option l'opzione da aggiungere
*/
public void addOption(String option)
{
}

/**
     Visualizza il menu sulla 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 pare più comodo. Il costruttore sembra 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
  - Un carattere di parentesi tonda chiusa
  - L'opzione da aggiungere
  - Un carattere speciale (`newline`) 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, dal momento che risulta essere 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 fa funzionare 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

## 3.6 Collaudo di unità

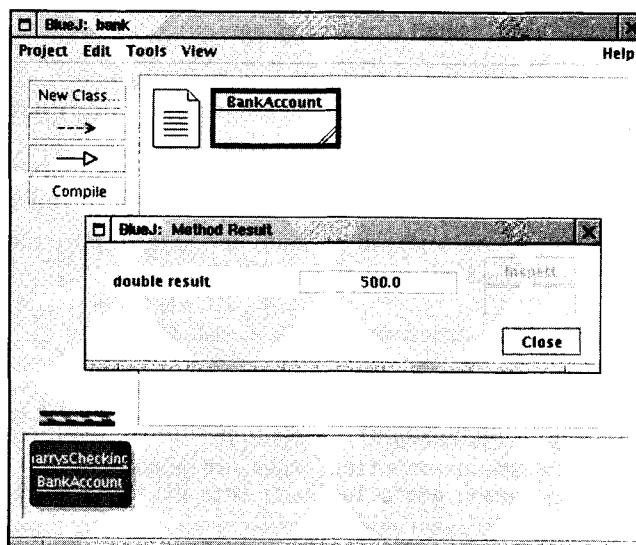
Nel paragrafo precedente abbiamo portato a termine la realizzazione della classe `BankAccount`. Come la si può usare? Potete sicuramente compilare il file `BankAccount.java`, ma non potete eseguire il file `BankAccount.class` che viene prodotto, perché non contiene un metodo `main`. Questa è una situazione molto comune: la maggior parte delle classi non contiene un metodo `main`.

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*.

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 avete visto in Argomenti avanzati 2.1), per cui potete collaudare una classe in modo molto semplice, costruendo con essa un oggetto, invocandone metodi e verificando che vengano prodotti i risultati attesi. La Figura 5 mostra il risultato dell'invocazione, all'interno di BlueJ, del metodo `getBalance` applicato a un oggetto della classe `BankAccount`.

**Figura 5**

valore restituito  
dal metodo `getBalance`,  
visualizzato con BlueJ



In alternativa, potete scrivere una classe per il collaudo (o *classe di test*). Una classe di test è una classe il cui metodo `main` contiene enunciati che servono al collaudo di un'altra classe. Come visto nel Paragrafo 2.9, una classe di test esegue solitamente questi passi:

1. Costruisce uno o più oggetti della classe che si sta collaudando.
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 2.9, è un valido esempio di classe di test: tale classe 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 ch03/account/BankAccountTester.java

```
/**
 * Una classe di collaudo per la classe BankAccount.
 */
public class BankAccountTester
{
    /**
     * Collauda i metodi della classe BankAccount.
     * @param args non utilizzato
     */
    public static void main(String[] args)
    {
        BankAccount harrysChecking = new BankAccount();
        harrysChecking.deposit(2000);
        harrysChecking.withdraw(500);
        System.out.println(harrysChecking.getBalance());
        System.out.println("Expected: 1500");
    }
}
```

### Esecuzione del programma

```
1500
Expected: 1500
```

Per ottenere un programma, occorre mettere insieme le due classi, `BankAccount` e `BankAccountTester`. I dettagli 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 vostro 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 sul campo un oggetto di tipo `BankAccount`.



### Auto-valutazione

16. Quando eseguite il programma `BankAccountTester`, quanti oggetti di tipo `BankAccount` vengono costruiti? E quanti oggetti di tipo `BankAccountTester`?
17. Perché la classe `BankAccountTester` non è necessaria negli ambienti di sviluppo che, come BlueJ, consentono il collaudo interattivo?

## 3.7 Variabili locali

**Le variabili locali sono dichiarate nel corpo di un metodo.**

In questo paragrafo parleremo del comportamento delle variabili *locali*. Una *variabile locale* 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 enterPayment(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*, “raccoglitrice di spazzatura”) che periodicamente elimina gli oggetti che non sono più utilizzati.

Le variabili di esemplare vengono inizializzate a un valore predefinito, mentre le variabili locali devono essere inizializzate esplicitamente.

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 la utilizzate 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 prima che venga invocato il costruttore. Le variabili di tipo numerico sono inizializzate a zero, mentre le variabili oggetto assumono il valore speciale `null`. Se un riferimento a oggetto ha il valore `null`, non fa riferimento ad alcun oggetto, come vedremo con maggior dettaglio nel Paragrafo 5.2.5.

## Auto-valutazione

18. Cosa hanno in comune le variabili locali e le variabili parametro? In quale aspetto essenziale sono invece diverse?
19. 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;`

## Errori comuni 3.2

### 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 mediante un riferimento di valore `null`.

Per evitare questo problema, prendete l'abitudine di inizializzare tutte le variabili di esemplare:

```
public BankAccount(double initialBalance)
{
    balance = initialBalance;
    owner = "None";
}
```

### 3.8 Parametri impliciti

Nel Paragrafo 2.4 avete appreso che un metodo ha un *parametro implicito* (l'oggetto con il quale viene invocato) e *parametri espliciti*, racchiusi tra parentesi tonde. In questo paragrafo approfondiremo il discorso sui parametri impliciti.

Osserviamo questa particolare invocazione del metodo `deposit`:

```
momsSavings.deposit(500);
```

Il parametro implicito è `momsSavings` e il parametro esplicito è `500`.

Analizziamo di nuovo il codice del metodo `deposit`:

```
public void deposit(double amount)
{
    balance = balance + amount;
}
```

Che significato ha, di preciso, `balance`? Dopo tutto, il nostro 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 rappresentare `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.

Se vi serve, potete accedere al parametro implicito (cioè all'oggetto con cui è stato invocato il metodo) usando la parola riservata `this`. Ad esempio, nella precedente invocazione del metodo, `this` faceva riferimento al medesimo oggetto a cui fa riferimento `momsSavings`, come si può vedere nella Figura 6.

L'enunciato

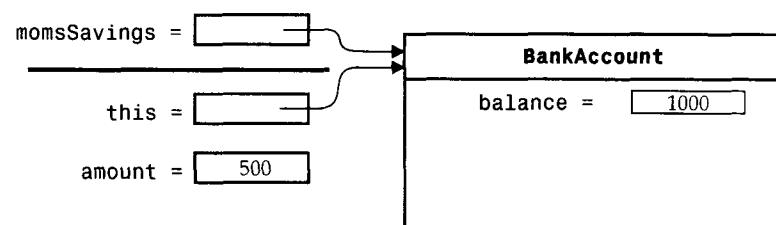
```
balance = balance + amount;
```

ha il seguente significato:

```
this.balance = this.balance + amount;
```

**Figura 6**

Il parametro implicito nell'invocazione di un metodo



Quando in un metodo ci si riferisce a una variabile di esemplare, il compilatore la associa automaticamente al parametro `this`. Alcuni programmatore preferiscono inserire manualmente il parametro `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. Esaminiamolo il costruttore:

```
public BankAccount(double balance)
{
    this.balance = balance;
}
```

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. In Java, quando si cerca il significato del nome di una variabile, si guardano prima le variabili locali e le variabili parametro. 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 aggiungiamo un metodo che applica il canone mensile del conto:

```
public class BankAccount
{
    ...
    public void monthlyFee()
    {
        withdraw(10); // preleva 10 dollari da questo conto
    }
}
```

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 (che non vediamo) del metodo `monthlyFee`.

Se pensate che un parametro implicito di questo tipo generi confusione, potete usare il riferimento `this`, per rendere il metodo più leggibile:

```
public class BankAccount
{
    ...
    public void monthlyFee()
    {
```

L'invocazione di un metodo priva di parametro implicito agisce sul medesimo oggetto su cui si sta operando.

```

        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

20. Quanti parametri impliciti e quanti parametri esplicativi ci sono nel metodo `withdraw` della classe `BankAccount`, e quali sono i loro nomi e tipi?
21. Qual è il significato di `this.amount` nel metodo `deposit`? Oppure, se l'espressione è priva di significato, per quale motivo lo è?
22. Quanti parametri impliciti e quanti parametri esplicativi ci sono nel metodo `main` della classe `BankAccountTester`, e quali sono i loro nomi?



## Argomenti avanzati 3.1

### Invocare un costruttore dall'interno di un altro costruttore

Esaminiamo la classe `BankAccount`, che ha due costruttori: un costruttore senza parametri, per inizializzare il saldo a zero, e un altro costruttore, per fornire un saldo iniziale. Invece di impostare esplicitamente il saldo a zero, un costruttore può invocare un altro costruttore della stessa classe. Per questa operazione, esiste una notazione abbreviata:

```

class BankAccount
{
    public BankAccount(double initialBalance)
    {
        balance = initialBalance;
    }

    public BankAccount()
    {
        this(0);
    }
    ...
}

```

L'enunciato `this(0)`; significa: "Invoca un altro costruttore di questa classe e fornisci il valore `0` come parametro di costruzione". Un'invocazione di costruttore di questo tipo può comparire solamente *quale prima riga di un altro costruttore*.

Questa sintassi rappresenta una comodità trascurabile, quindi non la useremo nel libro, perché in realtà l'utilizzo della parola riservata `this` genera un po' di confusione. Normalmente, `this` indica un riferimento al parametro implicito, ma, se `this` è seguito da parentesi tonde, rappresenta l'invocazione di un altro costruttore della stessa classe.

### 3.9 Classi per figure complesse

In questo paragrafo, proseguiamo la trattazione facoltativa degli argomenti grafici parlando di come organizzare in modo orientato agli oggetti il disegno di figure complesse.

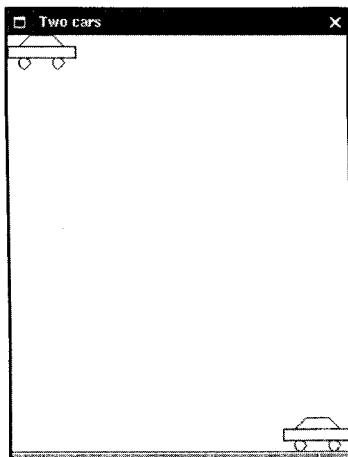
**Creare una classe per ogni porzione  
di figura che possa essere utilizzata  
più volte è un'ottima idea.**

Quando volete scrivere un programma che generi disegni articolati, composti di più parti, come quello di Figura 7, è bene creare una classe distinta per ogni porzione di figura. Predisponete un metodo che disegni la figura e un costruttore che ne definisca la posizione. Ecco, ad esempio, lo schema della classe `Car`, per disegnare un'automobile:

```
public class Car
{
    public Car(int x, int y)
    {
        // memorizza la posizione
        ...
    }
    public void draw(Graphics2D g2)
    {
        // istruzioni per il disegno
        ...
    }
}
```

**Figura 7**

Un componente che disegna due automobili



**Per capire come disegnare una figura complessa, fatene uno schizzo su carta millimetrata.**

Al termine del paragrafo troverete la dichiarazione completa della classe, nella quale noterete che il metodo `draw` contiene una lunga sequenza di istruzioni, per disegnare l'abitacolo, il tetto e le ruote. Le coordinate per le parti dell'automobile sembrano un po' casuali: per trovare i valori adatti, conviene disegnare l'immagine su carta millimetrata e rilevarne le coordinate, come in Figura 8.

Il programma che genera la Figura 7 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.



## Note di cronaca 3.1

### Apparati per il voto elettronico

Nelle elezioni presidenziali degli Stati Uniti tenutesi nel 2000 i voti sono stati raccolti da parecchi tipi di apparati diversi. 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 votazioni 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 aumentare la probabilità che ciascun voto venga contato seguendo esattamente le intenzioni di voto dell'elettore 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 commesso 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?

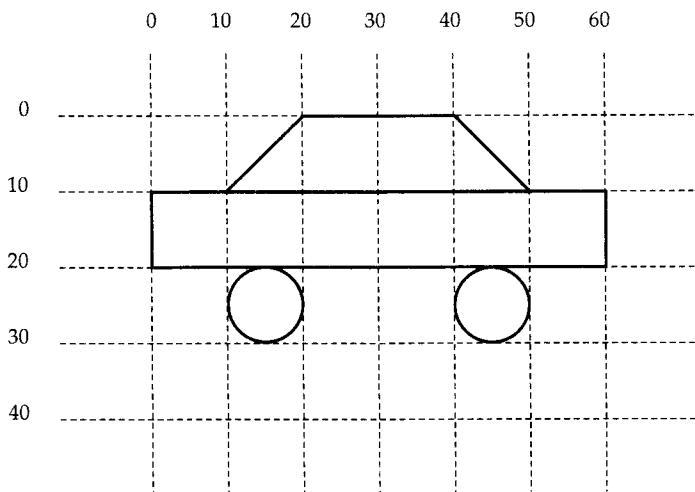
E, alla fine dei conti, la correttezza del funzionamento degli sportelli bancari automatici è più o meno

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.

**Figura 8**

Utilizzo di carta millimetrata per individuare le coordinate delle figure



- 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 la posizione in basso a destra, 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隐式的, 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 ch03/car/CarComponent.java

```
import java.awt.Graphics;
import java.awt.Graphics2D;
import javax.swing.JComponent;

/**
 * Questo componente disegna due automobili stilizzate.
 */
public class CarComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;

        Car car1 = new Car(0, 0);

        int x = getWidth() - 60;
        int y = getHeight() - 30;

        Car car2 = new Car(x, y);

        car1.draw(g2);
        car2.draw(g2);
    }
}
```

#### File ch03/car/Car.java

```
import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Line2D;
import java.awt.geom.Point2D;

/**
 * Una forma di automobile che può essere posizionata ovunque sullo schermo.
 */
public class Car
{
    private int xLeft;
    private int yTop;

    /**
     * Costruisce un'automobile con l'angolo in alto a sinistra assegnato.
     * @param x la coordinata x dell'angolo in alto a sinistra
     * @param y la coordinata y dell'angolo in alto a sinistra
     */
    public Car(int x, int y)
    {
```

```
        xLeft = x;
        yTop = y;
    }

    /**
     * Disegna l'automobile.
     * @param g2 il contesto grafico
     */
    public void draw(Graphics2D g2)
    {
        Rectangle body
            = new Rectangle(xLeft, yTop + 10, 60, 10);
        Ellipse2D.Double frontTire
            = new Ellipse2D.Double(xLeft + 10, yTop + 20, 10, 10);
        Ellipse2D.Double rearTire
            = new Ellipse2D.Double(xLeft + 40, yTop + 20, 10, 10);

        // la base del parabrezza
        Point2D.Double r1
            = new Point2D.Double(xLeft + 10, yTop + 10);
        // la parte anteriore del tettuccio
        Point2D.Double r2
            = new Point2D.Double(xLeft + 20, yTop);
        // la parte posteriore del tettuccio
        Point2D.Double r3
            = new Point2D.Double(xLeft + 40, yTop);
        // la base del lunotto posteriore
        Point2D.Double r4
            = new Point2D.Double(xLeft + 50, yTop + 10);

        Line2D.Double frontWindshield
            = new Line2D.Double(r1, r2);
        Line2D.Double roofTop
            = new Line2D.Double(r2, r3);
        Line2D.Double rearWindshield
            = new Line2D.Double(r3, r4);

        g2.draw(body);
        g2.draw(frontTire);
        g2.draw(rearTire);
        g2.draw(frontWindshield);
        g2.draw(roofTop);
        g2.draw(rearWindshield);
    }
}
```

### File ch03/car/CarViewer.java

```
import javax.swing.JFrame;

public class CarViewer
{
    public static void main(String[] args)
    {
        JFrame frame = new JFrame();
```

```

frame.setSize(300, 400);
frame.setTitle("Two cars");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

CarComponent component = new CarComponent();
frame.add(component);

frame.setVisible(true);
}
}

```



## Auto-valutazione

23. Quale classe si deve modificare per visualizzare due automobili affiancate?
24. Quale classe si deve modificare per disegnare automobili con le ruote nere, e quale modifica occorre fare?
25. Come si fa a disegnare automobili di dimensione doppia?



## Consigli pratici 3.2

### Disegnare forme grafiche

Si possono progettare programmi che visualizzino un'ampia varietà di forme grafiche: queste istruzioni delineano le fasi di una procedura per decomporre un disegno in parti, realizzando poi un programma che effettua il disegno completo. Useremo come esempio il disegno della bandiera italiana.

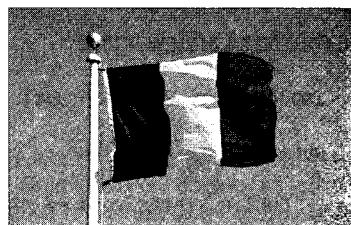
#### Fase 1 Determinate le forme che vi servono nel disegno completo

Potete usare le forme seguenti:

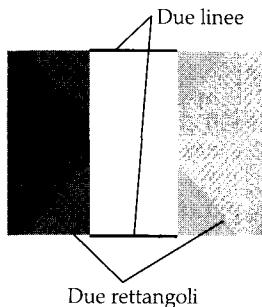
- Quadrati e rettangoli
- Cerchi ed ellissi
- Segmenti di linea retta (“linee”)

Si possono disegnare i profili di queste forme con qualsiasi colore, oppure se ne può colorare la parte interna con qualsiasi colore. Potete anche usare testo per etichettare parti del disegno.

Molte bandiere nazionali sono composte da tre sezioni 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 linee orizzontali nella parte centrale, in alto e in basso:



### Fase 2 Trovate le coordinate per le forme

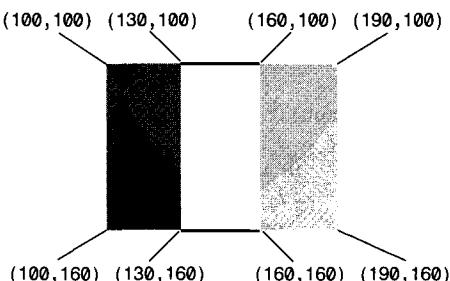
Ora occorre trovare le posizioni esatte per le singole figure geometriche.

- Per i rettangoli servono: le coordinate  $x$  e  $y$  dell'angolo superiore sinistro, la larghezza e l'altezza.
- Per le ellissi servono: l'angolo superiore sinistro, la larghezza e l'altezza del rettangolo che le delimita.
- Per le linee 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 della scritta da visualizzare.

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 siete in grado di calcolare le coordinate di tutti i punti significativi della figura:



**Fase 3** Scrivete gli enunciati Java per disegnare le forme

Nel nostro esempio ci sono due rettangoli e due linee:

```
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 quelle scelte. Se decidete di seguire l'approccio ambizioso, allora i rettangoli e le linee 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 dovete riempire i rettangoli e disegnare le linee. 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** Combinate gli enunciati di disegno con lo schema del componente

```
public class MyComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;
        // qui vanno inserite le istruzioni per disegnare
        ...
    }
}
```

Nel nostro esempio è possibile aggiungere, molto semplicemente, le istruzioni per disegnare tutte le forme direttamente nel metodo `paintComponent`:

```
public class ItalianFlagComponent extends JComponent
{
    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, specificando 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);
        ...
    }
}
```

Avete ancora bisogno di una classe separata per il componente, ma diventa molto semplice:

```
public class ItalianFlagComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;
```



## Note di cronaca 3.2

### La grafica al calcolatore

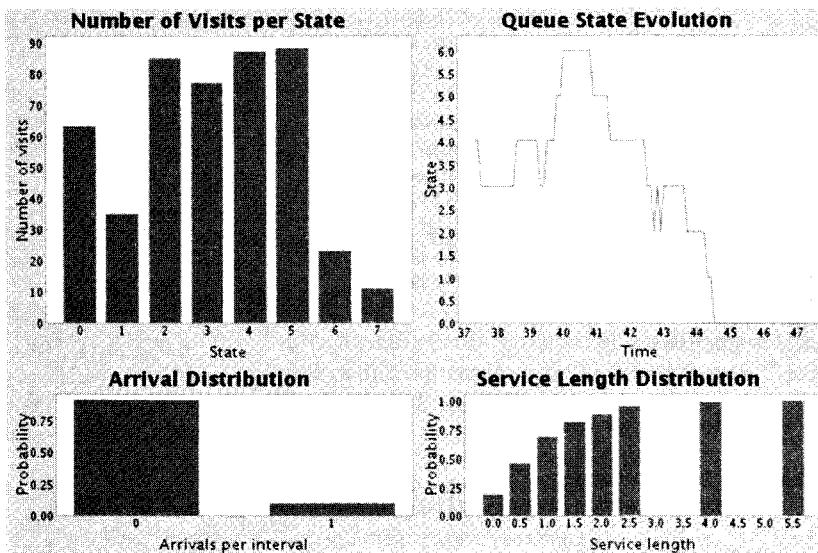
Generare e manipolare immagini è una delle applicazioni più stimolanti del computer. In questo campo, distinguiamo diversi tipi di *computer graphics*.

I *diagrammi*, come grafici numerici o mappe, sono strumenti che trasmettono informazioni al lettore: non descrivono direttamente nulla di quanto si verifica nel mondo naturale, ma sono uno strumento per visualizzare informazioni.

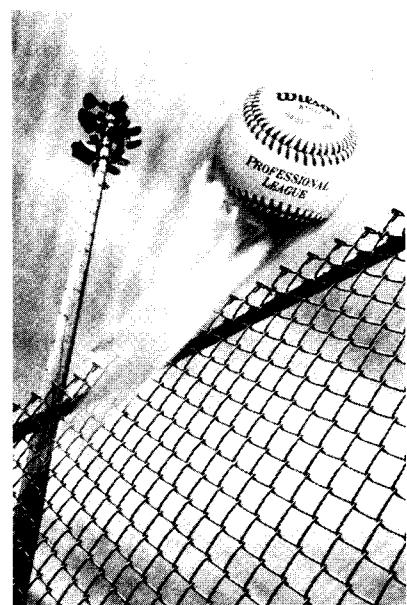
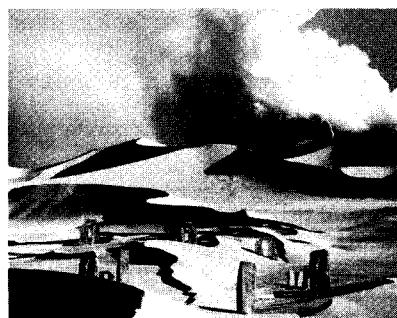
complesso e alquanto casuale. Il grado di realismo in queste immagini è in costante miglioramento.

Le *immagini manipolate* sono fotografie o sequenze cinematografiche di eventi reali, convertite in forma digitale e modificate al computer. Per esempio, le sequenze del film *Apollo 13* sono state prodotte cambiando la prospettiva di immagini reali, in modo da presentare il lancio del razzo da un punto di vista più suggestivo.

La grafica al calcolatore è uno dei campi più stimolanti dell'informatica. Richiede l'elaborazione di enormi quantità di informazioni a velocità elevatissima e, per questo, si inventano continuamente nuovi algoritmi. Visualizzare un insieme di oggetti tridimensionali sovrapposti, con profili curvilinei, richiede strumenti matematici avanzati. Per modellare realisticamente superfici e organismi biologici sono necessarie anche rilevanti conoscenze di matematica, di fisica e di biologia.



Le *scene* sono immagini generate al computer che tentano di rappresentare scene di un mondo reale o immaginario. Risulta piuttosto difficile rendere accuratamente luci e ombre. Occorre fare molti sforzi per evitare che le immagini appaiano eccessivamente piatte e semplici, poiché, nel mondo reale, nuvole, rocce, foglie e polvere hanno un aspetto



```

        ItalianFlag flag = new ItalianFlag(100, 100, 90);
        flag.draw(g2);
    }
}

```

#### Fase 5 Scrivete 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

### Le variabili di esemplare e i 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: cos'è e quali benefici porta

- 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 la localizzazione di errori e la modifica di dettagli realizzativi di una classe.

### Intestazioni di metodi e costruttori come descrizione dell'interfaccia pubblica di una classe

- Per realizzare una classe, occorre prima individuare quali metodi siano necessari.
- L'intestazione di un metodo ne specifica il tipo di dati restituito, il nome e i parametri (loro tipo e nome).
- I costruttori impostano i valori iniziali per i dati degli oggetti. Il nome di un costruttore è sempre uguale al nome della classe.

### Documentazione di una classe in formato javadoc

- Usate i commenti di documentazione per descrivere le classi e i metodi pubblici dei vostri programmi.
- Scrivete commenti di documentazione per ogni classe, ogni metodo, ogni parametro e ogni valore restituito.

### L'implementazione privata di una classe

- L'implementazione privata di una classe comprende le variabili di esemplare e il corpo di costruttori e metodi.
- Per costruire nuovi oggetti si usa l'operatore `new`, seguito dal nome di una classe e da parametri opportuni.

### Il collaudo che consente di verificare 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.

### 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.

### L'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 implicito.
- Il riferimento `this` rappresenta il parametro implicito.
- L'invocazione di un metodo privo di 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 ripasso

- \* **Esercizio R3.1.** Cos'è l'interfaccia pubblica di una classe? In cosa differisce dall'implementazione della classe stessa?
- \* **Esercizio R3.2.** Cos'è l'incapsulamento? Perché è utile?
- \* **Esercizio R3.3.** Le variabili di esemplare fanno parte dell'implementazione privata di una classe, ma sono visibili a quei programmatore che abbiano a disposizione il codice sorgente della classe stessa. Spiegiate in che senso la parola riservata `private` provvede a "nascondere" l'informazione.
- \* **Esercizio R3.4.** Considerate una classe `Grade` che rappresenti un voto, in lettere, come A+ o B. Delineate due diverse scelte per le variabili di esemplare da usare per realizzarla.
- \*\* **Esercizio R3.5.** Considerate una classe `Time` che rappresenti un istante di tempo, come le 9 del mattino o le 3.30 del pomeriggio. Delineate due diversi insiemi di variabili di esemplare che si potrebbero usare per realizzarla.

- ★ **Esercizio 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`?
- ★★ **Esercizio R3.7.** La variabile di esemplare `value` della classe `Counter` può essere ispezionata mediante il metodo d'accesso `getValue`. Ci dovrebbe essere un metodo `setValue` per modificare tale variabile? Motivate la vostra risposta.
- ★★ **Esercizio R3.8.**
  - a. 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?
  - b. Al contrario, potremmo togliere il costruttore `BankAccount()` e mettere a disposizione soltanto `BankAccount(double initialBalance)`?
- ★★ **Esercizio R3.9.** Perché la classe `BankAccount` non dispone di un metodo `reset`, che ne riporti lo stato al valore iniziale?
- ★ **Esercizio R3.10.** Cosa succede, nella nostra realizzazione della classe `BankAccount`, quando da un conto viene prelevata una quantità di denaro maggiore del saldo disponibile?
- ★★ **Esercizio R3.11.** Cos'è il riferimento `this`? Per quale motivo lo usereste?
- ★★ **Esercizio R3.12.** 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
}
```

- ★★ **Esercizio R3.13** Immaginate di voler realizzare una classe `TimeDepositAccount` che rappresenti un conto bancario dotato di un tasso di interesse 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 interessi 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.

- \* **Esercizio R3.14.** Analizzate questa classe:

```
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.

- \*\* Esercizio R3.15.** Analizzate questa classe:

```

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?

- \*\*\*T Esercizio R3.16.** Progettate una classe per il collaudo di unità della classe `Counter` vista nel Paragrafo 3.1.
- \*\*T Esercizio R3.17.** Leggete l'Esercizio P3.7, senza realizzare, 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.
- \*\*G Esercizio R3.18.** Nell'ipotesi di voler estendere il programma visualizzatore di automobili visto nel Paragrafo 3.9 in modo che mostri una scena urbana, con diverse automobili e case, di quali classi avreste bisogno?
- \*\*\*G Esercizio R3.19.** Spiegate per quale motivo nella classe `CarComponent` le invocazioni dei metodi `getWidth` e `getHeight` non hanno parametri espliciti.
- \*\*G Esercizio R3.20.** Come modifichereste la classe `Car` per poter visualizzare automobili di dimensioni diverse?

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

# 4

## Tipi di dati fondamentali

### Obiettivi del capitolo

- Apprendere l'utilizzo di numeri interi e di numeri in virgola mobile
- Identificare 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 in ingresso e a presentare dati impaginati in uscita

Questo capitolo insegna a manipolare numeri e sequenze di caratteri in Java, con l'obiettivo di raggiungere una salda conoscenza di tali tipi di dati fondamentali.

Apprenderete le proprietà e i limiti dei tipi di dati numerici in Java e vedrete come manipolare numeri e stringhe all'interno dei vostri programmi. Infine, tratteremo l'importante argomento dei dati in ingresso e in uscita, per consentirvi di realizzare programmi interattivi.

## 4.1 Tipi di numeri

**Java ha otto tipi primitivi, fra i quali quattro tipi interi e due tipi 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 tipi numerici, quattro dei quali rappresentano numeri interi e due rappresentano numeri in virgola mobile.

**Tabella 1**  
Tipi primitivi

Tipo	Descrizione	Dimensione
<code>int</code>	Tipo intero con intervallo $-2147483648\dots2147483647$ (circa 2 miliardi)	4 byte
<code>byte</code>	Tipo che descrive un singolo byte, con intervallo $-128\dots127$	1 byte
<code>short</code>	Tipo intero “corto”, con intervallo $-32768\dots32767$	2 byte
<code>long</code>	Tipo intero “lungo”, con intervallo $-9223372036854775808\dots9223372036854775807$	8 byte
<code>double</code>	Tipo in virgola mobile a doppia precisione, con intervallo circa $\pm10^{308}$ e circa 15 cifre decimali significative	8 byte
<code>float</code>	Tipo in virgola mobile a singola precisione, con intervallo circa $\pm10^{38}$ e circa 7 cifre decimali significative	4 byte
<code>char</code>	Tipo che rappresenta caratteri codificati secondo lo schema Unicode (Argomenti avanzati 4.5)	2 byte
<code>boolean</code>	Tipo per i due valori logici <code>true</code> e <code>false</code> (Capitolo 5)	1 bit

Ciascuno dei tipi interi ha un diverso intervallo di variabilità (la sezione Argomenti avanzati 4.2 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.

In generale, per le quantità intere useremo il tipo `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’intervallo di variabilità del tipo numerico, come in questo esempio:

```
int n = 1000000;
System.out.println(n * n); // Visualizza -727379968, ovviamente errato
```

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 incorrete in questo problema, la soluzione più semplice consiste nell’utilizzare il tipo `long`. La sezione Argomenti avanzati 4.1 vi mostrerà come usare, nella poco probabile circostanza in cui anche il tipo `long` dia luogo a trabocco, il tipo a precisione arbitraria `BigInteger`.

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  $\pm10^{308}$ .

**Un calcolo numerico trabocca se il risultato esce dall’intervallo del tipo numerico.**

e circa 15 cifre decimali significative, però è probabile che dobbiate evitare l'utilizzo del tipo `float`, che ha meno di 7 cifre significative (alcuni programmatori usano `float` per risparmiare memoria quando devono memorizzare un'enorme quantità di numeri e non hanno bisogno di molta precisione).

Quando si usano valori in virgola mobile si deve tener presente anche il più serio problema degli *errori di arrotondamento*, che possono avvenire durante la conversione tra numeri binari e numeri decimali oppure tra numeri interi e numeri in virgola mobile. Quando un valore non può essere convertito in modo esatto, viene arrotondato al valore rappresentabile più vicino, come in questo esempio:

```
double f = 4.35;
System.out.println(100 * f); // Visualizza 434.99999999999994
```

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 negli Argomenti avanzati 4.2).

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` (Argomenti avanzati 4.1).

In Java è perfettamente lecito assegnare un valore 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 4.3 vedrete come convertire un valore di tipo `double` in un valore intero.

## 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;`

**Avvengono errori di arrotondamento quando non è possibile fare una conversione numerica esatta.**





## 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`, gli “oggetti grandi numeri” non hanno sostanzialmente limiti di dimensione e di precisione. Tuttavia, i calcoli con tali oggetti sono molto più lenti di quelli che elaborano tipi numerici. Forse ancora più importante: con questi oggetti non è possibile utilizzare i consueti operatori aritmetici: al loro posto dovete invocare i metodi `add`, `subtract` e `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 1000000000000
```

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
```



## Argomenti avanzati 4.2

### Numeri binari

Tutti conoscete i numeri decimali, che usano le cifre 0, 1, 2, ..., 9. Ciascuna cifra ha un valore posizionale, pari a  $1 = 10^0$ ,  $10 = 10^1$ ,  $100 = 10^2$ ,  $1000 = 10^3$ , e così via. Per esempio:

$$435 = 4 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0$$

Le cifre frazionarie hanno valori posizionali correlati alle potenze negative di dieci:  $0.1 = 10^{-1}$ ,  $0.01 = 10^{-2}$ , e così via. Per esempio:

$$4.35 = 4 \cdot 10^0 + 3 \cdot 10^{-1} + 5 \cdot 10^{-2}$$

I computer usano invece numeri *binari*, composti da due sole cifre (0 e 1), con valori posizionali basati su potenze di 2. I numeri binari sono più facili da manipolare per i computer, perché è meno complicato costruire circuiti logici che distinguono fra “off” e “on”, piuttosto che circuiti in grado di discriminare fra dieci differenti livelli di voltaggio.

Trasformare in decimale un numero binario è facile: basta calcolare le potenze di due che corrispondono alle cifre “uno” presenti nel numero binario. Per esempio:

$$1101 \text{ binario} = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 1 = 13$$

I numeri binari frazionari usano potenze negative di due. Per esempio:

$$1.101 \text{ binario} = 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 1 + 0.5 + 0.125 = 1.625$$

Convertire numeri decimali in numeri binari è un po' più complicato. Ecco un algoritmo che converte un numero intero decimale nel suo equivalente binario: continuare a dividere il numero intero per 2, tenendo da parte i resti delle divisioni; fermatevi quando il numero diventa uguale a zero. Poi, scrivete sotto forma di numero binario la serie dei resti, iniziando dall'ultimo. Ecco un esempio:

$$\begin{aligned} 100 \div 2 &= 50 \text{ resto } \mathbf{0} \\ 50 \div 2 &= 25 \text{ resto } \mathbf{0} \\ 25 \div 2 &= 12 \text{ resto } \mathbf{1} \\ 12 \div 2 &= 6 \text{ resto } \mathbf{0} \\ 6 \div 2 &= 3 \text{ resto } \mathbf{0} \\ 3 \div 2 &= 1 \text{ resto } \mathbf{1} \\ 1 \div 2 &= 0 \text{ resto } \mathbf{1} \end{aligned}$$

Pertanto, 100 nel sistema decimale equivale a 1100100 in quello binario.

Per convertire un numero frazionario minore di 1 nel formato binario equivalente, continuare a moltiplicarlo per 2; ogni volta che il risultato è maggiore di 1, sottraete 1; fermatevi se il numero diventa uguale a zero. Poi, usate le cifre che precedono la virgola decimale nei risultati parziali quali cifre binarie della parte frazionaria, iniziando dalla prima. Per esempio:

$$\begin{aligned} 0.35 \cdot 2 &= \mathbf{0.7} \\ 0.7 \cdot 2 &= \mathbf{1.4} \\ 0.4 \cdot 2 &= \mathbf{0.8} \\ 0.8 \cdot 2 &= \mathbf{1.6} \\ 0.6 \cdot 2 &= \mathbf{1.2} \\ 0.2 \cdot 2 &= \mathbf{0.4} \end{aligned}$$

A questo punto lo schema si ripete, quindi la rappresentazione binaria di 0.35 è 0.01011001100110...

Per convertire in binario un numero in virgola mobile, convertite separatamente la parte intera e quella frazionaria. Per esempio, la rappresentazione binaria di 4.35 è 100.01011001100110...

Per programmare in Java non avete bisogno di conoscere a fondo i numeri binari, ma può essere utile saperne qualcosa. Per esempio, sapere che un valore di tipo `int` si rappresenta mediante un numero binario a 32 bit, spiega perché il numero intero più grande che si può utilizzare in Java è 0111 1111 1111 1111 1111 1111 1111 nella forma binaria, equivalente a 2147483647 in quella decimale (il primo bit è quello per il segno ed è zero per i valori positivi).

Per convertire un numero intero nella sua rappresentazione binaria potete usare il metodo statico `toString` della classe `Integer`. L'invocazione `Integer.toString(n, 2)` restituisce una stringa che contiene le cifre binarie del numero intero `n`. Viceversa, potete convertire in numero intero una stringa che contiene cifre binarie, invocando `Integer`.

`parseInt(digitString, 2)`. In entrambe queste invocazioni di metodi, il secondo parametro indica la base del sistema numerico e può essere qualsiasi numero compreso fra 2 e 36. Potete utilizzare questi due metodi per effettuare conversioni fra numeri inter decimali e binari. La libreria di Java non offre, però, metodi utili per fare la stessa operazione con numeri in virgola mobile.

Ora potete capire perché dobbiamo lottare con un errore di arrotondamento quando moltiplichiamo 4.35 per 100. Se eseguiamo la lunga moltiplicazione su carta, otteniamo:

```

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
1 0 0.0 1;0 1 1 0;0 1 1 0 ...
0
0
1 1 0 1 1 0 0 1 0.1 1 1 1 1 1 1 ...

```

In pratica, il risultato è 434, seguito da un numero infinito di cifre “uno”. La parte frazionaria del prodotto è l’equivalente binario di una frazione decimale infinita, 0.999999..., che è matematicamente uguale a 1, però la CPU può registrare solo un numero finito di cifre e ne elimina qualcuna quando converte il risultato in numero decimale.

## 4.2 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, 0.25, 0.1, 0.05 e 0.01, compaiono nell’espressione aritmetica senza alcuna spiegazione. Ovviamente in questo caso voi sapete che il valore di un “nickel” è cinque centesimi, giustificando così il numero 0.05, e così via, ma la persona che dopo di voi 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.



## Note di cronaca 4.1

### Errore nel calcolo in virgola mobile nel Pentium

Nel 1994, Intel Corporation rilasciò 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 del 1994, il Dott. Thomas Nicely del Lynchburg College, in Virginia, eseguì una lunga serie di calcoli per verificare le somme dei numeri 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 precedeva 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 (In-

stitute 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 (così come ne sono esenti le versioni successive di Pentium, come Pentium III e IV). 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 27 000 anni. Sfortunatamente per Intel, il Dott. Nicely non era un utente normale.

A questo punto, Intel si trovò a fronteggiare un problema reale: 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 che 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.

## 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

Dichiarata  
in un metodo.

La parola riservata `final` indica  
che questo valore non può essere  
modificato.

```
final double NICKEL_VALUE = 0.05;
```

Nei nomi delle costanti usate  
soltanto lettere maiuscole.

Dichiarata  
in una classe.

```
public static final double LITERS_PER_GALLON = 3.785;
```

**Una variabile `final` è una costante:  
dopo che le è stato assegnato un  
valore, non può più essere modificata.**

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 mai 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 composti da 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. 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 la 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;
```

**Date un nome alle costanti, per  
rendere i vostri programmi più facili  
da leggere e da modificare.**

Spesso le costanti vengono usate in più metodi, per cui è comodo dichiararle insieme alle variabili di esemplare della classe, identificandole come `static`, oltre che `final`. Come nel caso precedente, `final` significa che il valore è costante. 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;
}

```

In qualsiasi 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 cui 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 a disposizione un metodo che ha come valori di ingresso *conteggi di monete*. Ad esempio, l'invocazione

```
register.enterPayment(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 “nickel” e quattro “penny”. Il metodo `enterPayment` calcola l’importo totale pagato, 1.69 dollari. Come potete vedere esaminando il codice, il metodo usa costanti per rappresentare i valori delle singole monete.

## File ch04/cashregister/CashRegister.java

```
/**  
 * Un registratore di cassa somma gli articoli venduti  
 * e calcola il resto dovuto al cliente.  
 */  
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;  
    }  
  
    /**  
     * Registra la quantità di denaro pagata dal cliente.  
     * @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 nickel  
     * @param pennies il numero di penny  
     */  
    public void enterPayment(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 azzera la macchina per  
     * la vendita successiva.  
     * @return il resto dovuto al cliente  
     */  
    public double giveChange()  
{
```

```

        double change = payment - purchase;
        purchase = 0;
        payment = 0;
        return change;
    }
}

```

### File ch04/cashregister/CashRegisterTester.java

```

/*
 * Questa classe collauda la classe CashRegister.
 */
public class CashRegisterTester
{
    public static void main(String[] args)
    {
        CashRegister register = new CashRegister();

        register.recordPurchase(0.75);
        register.recordPurchase(1.50);
        register.enterPayment(2, 0, 5, 0, 0);
        System.out.print("Change: ");
        System.out.println(register.giveChange());
        System.out.println("Expected: 0.25");

        register.recordPurchase(2.25);
        register.recordPurchase(19.25);
        register.enterPayment(23, 2, 0, 0, 0);
        System.out.print("Change: ");
        System.out.println(register.giveChange());
        System.out.println("Expected: 2.0");
    }
}

```

### Esecuzione del programma

```

Change: 0.25
Expected: 0.25
Change: 2.0
Expected: 2.0

```

### Auto-valutazione

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;

```



## Consigli per la qualità 4.1

### Non usate 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 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 in una stringa in modo che stringhe diverse generino con buona probabilità codici di hash diversi: il valore 31 ha l’effetto di “mescolare” in modo appropriato 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 dovreste mai usare numeri magici nel codice: qualsiasi numero che non sia completamente auto-espli cativo 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;
```

Al contrario, un’abitudine come questa:

```
final int THREE_HUNDRED_AND_SIXTY_FIVE = 365;
```

è controproducente e da biasimare.

## 4.3 Operazioni aritmetiche e funzioni matematiche

In questo paragrafo vedrete come eseguire calcoli aritmetici in Java.

### 4.3.1 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 si usa l’operatore \*. La divisione si indica con /, non tramite il simbolo della frazione. Per esempio

$$\frac{a+b}{2}$$

diventa

$$(a + b) / 2$$

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 viene divisa per 2. Al contrario, nell'espressione:

$$a + b / 2$$

soltanto  $b$  è diviso per 2, poi viene calcolata la somma di  $a$  e di  $b / 2$ . 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.

### 4.3.2 Incremento e decremento

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:

```
items++;
```

~~gli operatori ++ e -- aumentano e diminuiscono di un'unità il valore di una variabile.~~

Questo enunciato somma 1 alla variabile `items`, ma è più pratico da digitare e da leggere rispetto all'enunciato di assegnamento equivalente:

```
items = items + 1;
```

Come forse avrete indovinato, esiste anche un operatore per il decremento, `--`. L'enunciato seguente sottrae 1 alla variabile `items`:

```
items--;
```

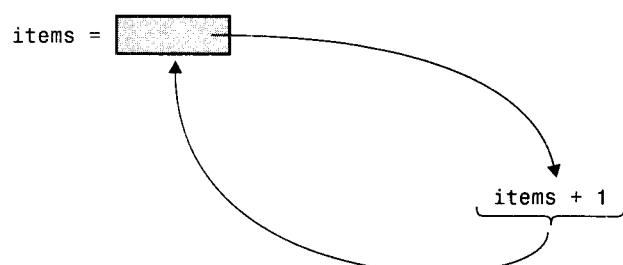
### 4.3.3 Divisione intera

~~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 restituiscono 1.75:

**Figura 1**

Incrementare una variabile



```
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. 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 (consultate Errori comuni 4.1).

Se interessa solamente il resto della divisione, si usa l'operatore %. Il risultato calcolato per l'espressione

```
7 % 4
```

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.

Vediamo un tipico utilizzo della divisione intera / e di calcolo del resto mediante l'operatore %. Supponiamo di voler conoscere il valore delle monete presenti in un borsellino, espresso separatamente in dollari e centesimi. Possiamo calcolare il valore tramite un numero intero che conteggia i centesimi, quindi risalire al numero di dollari interi e ai centesimi rimanenti.

```
final int PENNIES_PER_NICKEL = 5;
final int PENNIES_PER_DIME = 10;
final int PENNIES_PER_QUARTER = 25;
final int PENNIES_PER_DOLLAR = 100;

// calcola il valore totale in centesimi
int total = dollars * PENNIES_PER_DOLLAR + quarters * PENNIES_PER_QUARTER
           + nickels * PENNIES_PER_NICKEL * dimes * PENNIES_PER_DIME + pennies;

// usa la divisione intera per convertire in dollari e centesimi
int dollars = total / PENNIES_PER_DOLLAR;
int cents = total % PENNIES_PER_DOLLAR;
```

Per esempio, se total fosse 243, allora dollars varrebbe 2 e cents 43.

### 4.3.4

### Potenze e radici

La classe `Math` contiene i metodi `sqrt` e `pow` per calcolare radici quadrate e potenze.

Per calcolare  $x^n$  si usa il metodo `Math.pow(x, n)` (ma per calcolare  $x^2$  è molto più efficiente scrivere semplicemente `x * x`).

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)`.

In algebra si usano frazioni, esponenti e segni di estrazione di radice per comporre espressioni in una forma bidimensionale compatta, mentre in Java è utile scrivere tutte le espressioni secondo una disposizione lineare. Per esempio, questa è l'escissione della formula dell'equazione di secondo grado:

L'operatore % calcola il resto  
di una divisione.

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

in Java, diventa:

```
(-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a)
```

La Figura 2 mostra come analizzare un'espressione simile. Nelle espressioni complicate come questa, talvolta non è facile conservare la corrispondenza delle parentesi: consultate gli Errori comuni 4.2.

La Tabella 2 riporta ulteriori metodi della classe `Math`. In questi metodi, i valori dei parametri e i valori restituiti sono in virgola mobile.

**Figura 2**

Analisi di un'espressione

$$\begin{aligned}
 & (-b + \underbrace{\mathbf{Math.sqrt(b * b - 4 * a * c))}}_{\substack{b^2 \\ 4ac}} \underbrace{/}_{2a} \\
 & \quad \underbrace{\mathbf{b^2 - 4ac}}_{\substack{\sqrt{b^2 - 4ac}}} \\
 & \quad \underbrace{\mathbf{-b + \sqrt{b^2 - 4ac}}}_{\substack{-b + \sqrt{b^2 - 4ac}}} \\
 & \quad \underbrace{\mathbf{-b + \sqrt{b^2 - 4ac}}}_{2a}
 \end{aligned}$$

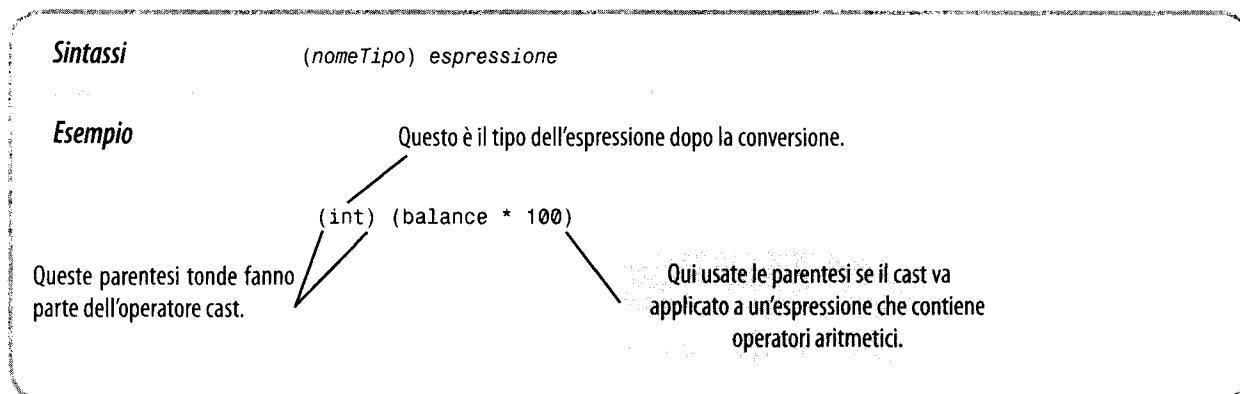
**Tabella 2**

Metodi matematici

<code>Math.sqrt(x)</code>	radice quadrata di $x$ ( $\geq 0$ )
<code>Math.pow(x, y)</code>	$x^y$ ( $x > 0$ , oppure $x = 0$ e $y > 0$ , oppure $x < 0$ e $y$ è intero)
<code>Math.sin(x)</code>	seno di $x$ ( $x$ in radianti)
<code>Math.cos(x)</code>	coseno di $x$
<code>Math.tan(x)</code>	tangente di $x$
<code>Math.asin(x)</code>	arcoseno ( $\sin^{-1} x \in [-\pi/2, \pi/2]$ , $x \in [-1, 1]$ )
<code>Math.acos(x)</code>	arcocoseno ( $\cos^{-1} x \in [0, \pi]$ , $x \in [-1, 1]$ )
<code>Math.atan(x)</code>	arcotangente ( $\tan^{-1} x \in [-\pi/2, \pi/2]$ )
<code>Math.atan2(y, x)</code>	arcotangente ( $\tan^{-1}(y/x) \in [-\pi, \pi]$ , $x$ può essere 0)
<code>Math.toRadians(x)</code>	converte $x$ da gradi a radianti (cioè restituisce $x \cdot \pi/180$ )
<code>Math.toDegrees(x)</code>	converte $x$ da radianti a gradi (cioè restituisce $x \cdot 180/\pi$ )
<code>Math.exp(x)</code>	$e^x$
<code>Math.log(x)</code>	logaritmo naturale ( $\ln(x)$ , $x > 0$ )
<code>Math.log10(x)</code>	logaritmo decimale ( $\log_{10}(x)$ , $x > 0$ )
<code>Math.round(x)</code>	il numero intero (di tipo long) più prossimo a $x$
<code>Math.ceil(x)</code>	il più piccolo numero intero (di tipo double) che sia $\geq x$
<code>Math.floor(x)</code>	il più grande numero intero (di tipo double) che sia $\leq x$
<code>Math.abs(x)</code>	valore assoluto, $ x $
<code>Math.max(x, y)</code>	il numero maggiore tra $x$ e $y$
<code>Math.min(x, y)</code>	il numero minore tra $x$ e $y$

## Sintassi di Java

### 4.2 Cast



#### 4.3.5 Conversione e arrotondamento

A volte avete un valore di tipo `double` e lo dovete convertire in un valore di tipo `int`. Per risolvere questo problema, si usa l'operatore `cast` ("forzatura"), che 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;
```

Per convertire un valore in un tipo diverso si usa il cast (*nomeTipo*).

Per arrotondare un numero in virgola mobile all'intero più vicino usate il metodo `Math.round`.

Il cast `(int)` converte in 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.

Il cast informa il compilatore della vostra consapevolezza di un'eventuale *perdita di informazione*: in questo caso, della perdita della parte frazionaria del numero. Si possono effettuare conversioni con cast anche per ottenere tipi di dati diversi, come `(float)` oppure `(byte)`.

Se volete, invece, arrotondare un numero in virgola mobile all'intero più vicino, usate il metodo `Math.round`, che restituisce un intero di tipo `long`, perché numeri in virgola mobile di valore elevato non possono essere memorizzati come `int`.

```
long rounded = Math.round(balance);
```

Se `balance` vale 13.75, a `rounded` viene assegnato il valore 14.



#### Auto-valutazione

6. Qual è il valore di `n` dopo l'esecuzione di questa sequenza di enunciati?  
`n--;`  
`n++;`  
`n--;`
7. Qual è il valore dell'espressione `1729 / 100`? E quello di `1729 % 100`?
8. Perché l'enunciato seguente non calcola il valore della media tra `s1`, `s2` e `s3`?

```
double average = s1 + s2 + s3 / 3; // Errore
```

**Tabella 3**  
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>	In Java, le formule complesse vengono "appiattite".
$\sqrt{a^2 + b^2}$	<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.

9. Come si esprime in notazione matematica l'espressione `Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2))`?
10. In quali situazioni il cast (`long`)  $x$  produce un risultato diverso dall'invocazione `Math.round(x)`?
11. In che modo potete arrotondare al più vicino valore di tipo `int` il valore  $x$  di tipo `double`, sapendo che è inferiore a  $2 \times 10^9$ ?

## Errori comuni 4.1

### Divisione fra numeri interi

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 s1 = 5; // punteggio della prova 1
int s2 = 6; // punteggio della prova 2
int s3 = 3; // punteggio della prova 3
double average = (s1 + s2 + s3) / 3; // Errore
System.out.print("Your average score is ");
System.out.println(average);
```

Che cosa può esservi di sbagliato? Ovviamente, la media di `s1, s2` e `s3` è:

$$\frac{s_1 + s_2 + s_3}{3}$$

Tuttavia, nell'esempio la barra `/` non indica una divisione nel senso matematico, ma indica una divisione fra numeri interi, perché sia `s1 + s2 + s3` sia `3` sono valori interi. Dato che la somma dei punteggi qui usati è 14, la media restituita è 4, vale a dire il risultato

della divisione intera di 14 per 3. In conclusione, nella variabile in virgola mobile `average` viene inserito il numero intero 4. La soluzione consiste nell'utilizzare un numeratore e un denominatore in virgola mobile:

```
double total = s1 + s2 + s3;
double average = total / 3;
```

oppure:

```
double average = (s1 + s2 + s3) / 3.0;
```

## Errori comuni 4.2

### Parentesi non accoppiate

Esaminiamo l'espressione:

```
1.5 * ((-b - Math.sqrt(b * b - 4 * a * c)) / (2 * a))
```

Cosa c'è che non va? Contate le parentesi: ci sono cinque parentesi aperte, ma soltanto quattro 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:

```
1.5 * (Math.sqrt(b * b - 4 * a * c)) - ((b / (2 * a))
```

L'espressione presenta cinque parentesi aperte e cinque parentesi chiuse, ma di nuovo non è corretta. Nella parte centrale dell'espressione ci sono soltanto due parentesi aperte ma tre 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 controlate 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 arriva a zero, le parentesi non sono bilanciate. Per esempio, se scorrete l'espressione precedente, dovreste mormorare:

```
1.5 * (Math.sqrt(b * b - 4 * a * c) ) ) / ((b / (2 * a))
      1       2           1 0 -1
```

e, quindi, trovare l'errore.

## Consigli per la qualità 4.2

### Spaziature

Al compilatore non importa se scrivete tutto il programma in un'unica riga o se inserite ciascun simbolo in una riga a sé stante. Al lettore umano, invece, interessa molto. Dovreste

usare righe vuote per raggruppare visivamente il vostro codice in sezioni distinte: nei listati del codice sorgente in questo libro troverete numerosi esempi di questo tipo.

Anche gli spazi vuoti all'interno delle espressioni sono importanti. 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 + - \* / %. Tuttavia, non inserite 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`. Non inserite spazi fra il nome di un metodo e le sue parentesi, ma inserite uno spazio dopo ciascuna parola riservata di Java: sarà così più facile capire che `sqrt`, in `Math.sqrt(x)`, è il nome di un metodo, mentre la parola `if`, in `if (x > 0)`, è una parola riservata.

## Consigli per la qualità 4.3

### Identificare nel codice le componenti comuni

Supponiamo di voler calcolare entrambe le soluzioni dell'equazione di secondo grado  $ax^2 + bx + c = 0$ . La formula dell'equazione di secondo grado indica che le soluzioni sono:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In Java, non esiste l'operatore `±`, che indica come ottenere due soluzioni simultaneamente, e bisogna risolvere le due espressioni separatamente:

```
x1 = (-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a);
x2 = (-b - Math.sqrt(b * b - 4 * a * c)) / (2 * a);
```

Questo schema presenta due problemi. Per prima cosa, il calcolo di `Math.sqrt(b * b - 4 * a * c)` viene eseguito due volte, con conseguente perdita di tempo. Inoltre, ogni volta che si replica lo stesso codice, aumenta la possibilità di errori di battitura. La soluzione consiste nell'estrare le *componenti comuni* presenti nel codice:

```
double root = Math.sqrt(b * b - 4 * a * c);
x1 = (-b + root) / (2 * a);
x2 = (-b - root) / (2 * a);
```

È possibile andare anche oltre ed estrarre il fattore comune `2 * a`, ma i vantaggi di questa tecnica, quando si tratta di calcoli molto semplici, sono limitati: il codice risultante può essere di scarsa leggibilità.



## Errori comuni 4.3

### Errori di arrotondamento

Quando si fanno calcoli con numeri in virgola mobile, gli errori di arrotondamento sono naturali. Probabilmente vi siete già imbattuti in questo fenomeno, nel corso di calcoli manuali: Per esempio, se calcolate  $1/3$  con due cifre decimali, ottenete 0.33 e, quando moltiplicate nuovamente per 3, il risultato è 0.99, non 1.00.

All'interno dei processori, i numeri sono rappresentati nel sistema numerico binario e non in quello decimale: trascurando cifre binarie riscontrerete ancora errori di arrotondamento, semplicemente questi possono presentarsi in posizioni diverse da quelle che ci si aspetta. Ecco un esempio:

```
double f = 4.35;
int n = (int) (100 * f);
System.out.println(n); // visualizza 434!
```

Naturalmente, il risultato di 4.35 moltiplicato per 100 è 435, ma il programma visualizza 434.

I computer rappresentano i numeri nel sistema binario (come visto in Argomenti avanzati 4.2). In questo sistema numerico non esiste una rappresentazione esatta per 4.35, proprio come la rappresentazione esatta di  $1/3$  è assente nel sistema decimale. La rappresentazione utilizzata dal computer ha un valore appena inferiore a 4.35, quindi, quando la si moltiplica per 100, il risultato è un po' inferiore a 435. Quando si converte un numero in virgola mobile in un intero, la parte frazionaria viene completamente scartata, anche se è prossima a 1: di conseguenza, nella variabile `n` viene memorizzato il numero intero 434. La soluzione consiste nell'adoperare `Math.round` per convertire in interi i numeri in virgola mobile:

```
int n = (int) Math.round(100 * f); // Va bene: n vale 435
```



## Argomenti avanzati 4.3

### Combinare assegnazioni e operatori aritmetici

In Java, possiamo combinare operatori aritmetici e assegnazioni. Per esempio, l'enunciato:

```
balance += amount;
```

è un'abbreviazione per:

```
balance = balance + amount;
```

Analogamente,

```
items *= 2;
```

è un modo alternativo per scrivere:

```
items = items * 2;
```

Molti programmatore 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.

## 4.4 Invocare metodi statici

Nel paragrafo precedente avete visto la classe `Math`, che contiene una raccolta di metodi utili per eseguire calcoli matematici. Questi metodi hanno una forma un po' particolare: sono *metodi statici*, che non agiscono su alcun oggetto.

Ciò significa che non si può scrivere:

```
double root = 100.sqrt(); // Errore
```

Il motivo è che, in Java, i numeri non sono oggetti, per cui non è possibile invocare un metodo applicandolo a un numero. Diversamente, si fornisce il numero a un metodo sotto la forma di parametro esplicito, racchiudendolo quindi fra parentesi dopo il nome del metodo:

```
double root = Math.sqrt(100);
```

Un metodo statico non agisce  
su un oggetto.

In questa invocazione sembra che il metodo `sqrt` sia applicato a un oggetto di nome `Math`, però `Math` è una classe, non un oggetto. Un metodo come `Math.sqrt`, che non agisce su alcun oggetto, viene qualificato come *static* (il termine “statico” proviene, storicamente, dai linguaggi C e C++ e non ha niente a che vedere con il significato consueto della parola stessa). Al contrario, un metodo che viene invocato usando un oggetto si chiama *metodo di esemplare*:

```
harrysChecking.deposit(100); // deposit è un metodo di esemplare
```

I metodi statici non agiscono su oggetti, ma sono comunque definiti all'interno di classi: quando invocate il metodo `sqrt`, dovete specificare la classe a cui esso appartiene, scrivendo `Math.sqrt(x)`.

Come si fa a capire che `Math` è una classe e non un oggetto? Per convenzione, i nomi delle classi iniziano con una lettera maiuscola (come `Math` o `BankAccount`), mentre i nomi di oggetti e di metodi iniziano con una lettera minuscola (come `harrysChecking` e `println`). Quindi, `harrysChecking.deposit(100)` indica l'invocazione del metodo `deposit` sull'oggetto `harrysChecking` di tipo `BankAccount`. Al contrario, `Math.sqrt(x)` indica l'invocazione del metodo `sqrt` della classe `Math`.

Questo utilizzo di lettere maiuscole e minuscole è soltanto una convenzione e non una regola del linguaggio Java, ma è una convenzione che gli autori delle classi della libreria di Java seguono rigidamente e dovreste fare la stessa cosa nei vostri programmi, in modo da non confondere i vostri colleghi programmatore.

### Auto-valutazione

12. Perché non si può invocare `x.pow(y)` per calcolare  $x^y$ ?
13. `System.out.println(4)` è l'invocazione di un metodo statico?



## Sintassi di Java

### 4.3 Invocazione di metodo statico

<b>Sintassi</b>	<code>NomeClasse.nomeMetodo(parametri)</code>
<b>Esempio</b>	<p>La classe in cui si trova la dichiarazione del metodo <code>pow.</code></p> <p>Tutti i parametri di un metodo statico sono espliciti.</p> <p><code>Math.pow(10, 3)</code></p>



### Consigli pratici 4.1

#### Effettuare calcoli

Molti problemi di programmazione richiedono l'utilizzo di formule matematiche per calcolare valori, ma non sempre è ovvio come trasformare l'enunciazione di un problema in una serie di formule matematiche e, infine, in enunciati nel linguaggio di programmazione Java.

**Fase 1** Analizzate il problema. Quali sono i dati in ingresso? Quali sono i dati da produrre?

Ad esempio, supponiamo che vi sia chiesto di simulare una macchina per la vendita di francobolli. Un cliente inserisce monete nella macchina, poi preme il pulsante “First class stamps” (*francobolli per posta prioritaria*); la macchina fornisce un numero di francobolli che corrisponde all’importo pagato dal cliente (un francobollo di quel tipo costava 44 centesimi di dollaro nel momento in cui il libro è stato scritto); infine, il cliente preme il pulsante “Penny stamps” (*francobolli da un centesimo*) e la macchina fornisce il resto in francobolli da un penny.

In questo problema c’è un solo dato in ingresso:

- La quantità di denaro inserita dal cliente

e ci sono due risultati da produrre:

- Il numero di francobolli di tipo “first class” che la macchina deve fornire
- Il numero di francobolli da un penny che la macchina deve fornire

**Fase 2** Risolvete 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 francobollo di tipo “first class” costi 44 centesimi di dollaro e che il cliente inserisca un dollaro, che è sufficiente per due francobolli (88 centesimi) ma

non è abbastanza per 3 francobolli (\$ 1.32). Di conseguenza, la macchina restituisce due francobolli di tipo “first class” e 12 francobolli da un penny.

### Fase 3 Progettate una classe che svolga i calcoli

Nella sezione Consigli pratici 3.1 avete visto come progettare una classe, identificandone i metodi e le variabili di esemplare. In questo caso, l'enunciazione del problema descrive tre metodi:

- `public void insert(int dollars)`
- `public int giveFirstClassStamps()`
- `public int givePennyStamps()`

Più difficile è la determinazione delle variabili di esemplare che descrivono lo stato della macchina: in questo esempio, una scelta ottima è rappresentata da un'unica variabile, che memorizzi il saldo disponibile, istante per istante (nell'Esercizio P4.12 potete vedere una scelta diversa).

Il saldo così descritto aumenta per effetto dell'invocazione del metodo `insert` e diminuisce durante l'esecuzione dei metodi `giveFirstClassStamps` e `givePennyStamps`.

### Fase 4 Scrivete pseudocodice che descriva i metodi

Data una somma di denaro e il prezzo di un francobollo di tipo “first class”, come si può calcolare quanti francobolli di quel tipo si possono comprare con tale somma? La risposta è ovviamente in relazione con il quoziente

$$\frac{\text{Somma di denaro}}{\text{Prezzo di un francobollo}}$$

Ad esempio, supponiamo che il cliente paghi un dollaro. Usando una calcolatrice tascabile calcoliamo il quoziente:  $\$ 1.00 / \$ 0.44 \approx 2.27$ .

Come si fa a ottenere la risposta “2 francobolli” dal numero 2.27? È la sua parte intera, che è facilmente calcolabile in Java quando i due argomenti da elaborare sono numeri interi; passiamo quindi a fare calcoli in centesimi, ottenendo:

$$\text{numero di francobolli “first class”} = 100 / 44 \text{ (divisione intera, senza resto)}$$

Cosa succede se l'utente inserisce due dollari? Il numeratore diventa 200. Cosa succede se il prezzo dei francobolli aumenta? Nel caso più generale, il calcolo da eseguire è questo:

$$\text{denaro inserito in centesimi} = 100 \times \text{denaro inserito in dollari}$$

$$\text{numero di francobolli “first class”} =$$

$$\text{denaro inserito in centesimi} / \text{prezzo di un francobollo “first class” in centesimi} \\ (\text{divisione intera, senza resto})$$

Quanti soldi rimangono dopo l'emissione dei francobolli di tipo “first class”? Ecco un modo per calcolarlo: dopo che il cliente ha avuto i francobolli “first class”, il denaro rimasto è il totale originario diminuito del valore dei francobolli acquistati. Nel nostro

esempio, il resto è 12 centesimi, cioè la differenza tra 100 e  $2 \times 44$ . Ecco, quindi, la formula generale:

$$\text{denaro rimasto} = \text{denaro inserito in centesimi} - \text{numero di francobolli "first class"} \times \\ \text{prezzo di un francobollo "first class" in centesimi}$$

### Fase 5 Realizzate la classe

Nella Fase 3 abbiamo deciso che lo stato della macchina può essere rappresentato dal saldo disponibile istante per istante e nella Fase 4 abbiamo capito che è meglio rappresentare tale saldo in centesimi.

In generale, è utile riscrivere lo pseudocodice usando esplicitamente le variabili di esemplare così individuate: nel nostro caso, sostituiamo **balance** al posto di quello che avevamo chiamato **denaro inserito in centesimi**. Quando si inseriscono monete, il saldo aumenta:

```
balance = balance + 100 * dollars;
```

Emettendo francobolli di tipo "first class", il saldo diminuisce, in questo modo:

```
firstClassStamps = balance / FIRST_CLASS_STAMP_PRICE;
balance = balance - firstClassStamps * FIRST_CLASS_STAMP_PRICE;
```

La quantità che avevamo chiamato **denaro rimasto** è ora, semplicemente, il valore della variabile di esemplare **balance**.

Ecco la realizzazione della classe **StampMachine**:

```
public class StampMachine
{
    public static final double FIRST_CLASS_STAMP_PRICE = 44;
    private int balance;

    public StampMachine()
    {
        balance = 0;
    }

    public void insert(int dollars)
    {
        balance = balance + 100 * dollars;
    }

    public int getFirstClassStamps()
    {
        int firstClassStamps = balance / FIRST_CLASS_STAMP_PRICE;
        balance = balance - firstClassStamps * FIRST_CLASS_STAMP_PRICE;
        return firstClassStamps;
    }

    public int givePennyStamps()
    {
        int pennyStamps = balance;
        balance = 0;
    }
}
```

```

        return pennyStamps;
    }
}

```

#### Fase 6 Collaudate la vostra classe

Eseguite un programma di collaudo (o usate un ambiente integrato, come BlueJ) per verificare che i valori calcolati dalla vostra classe siano uguali a quelli che potete calcolare a mano.

Ecco un programma di collaudo:

```

public class StampMachineTester
{
    public static void main(String[] args)
    {
        StampMachine machine = new StampMachine();
        machine.insert(1);
        System.out.print("First class stamps: ");
        System.out.println(machine.giveFirstClassStamps());
        System.out.println("Expected: 2");
        System.out.print("Penny stamps: ");
        System.out.println(machine.givePennyStamps());
        System.out.println("Expected: 12");
    }
}

```

#### Esecuzione del programma

```

First class stamps: 2
Expected: 2
Penny stamps: 12
Expected: 12

```

## Esempi completi 4.1

### Calcolare il volume e l'area della superficie totale di una piramide



Immaginate di voler essere d'aiuto agli archeologi che fanno ricerche sulle piramidi egiziane e di esservi assunti il compito di scrivere un metodo che determini il volume e l'area della superficie totale di una piramide, di cui sia nota la misura della base e dell'altezza.

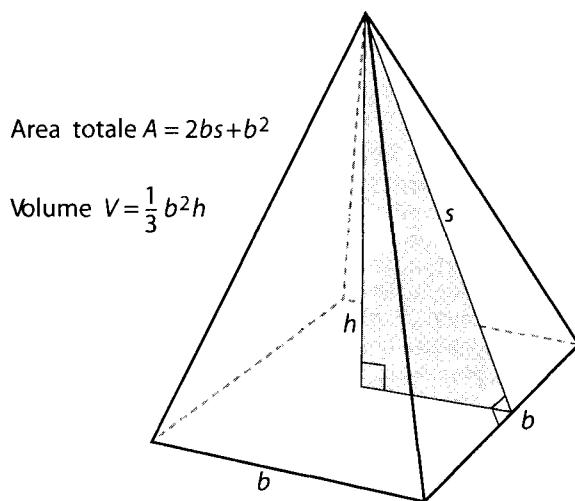
#### Fase 1 Analizzate il problema. Quali sono i dati in ingresso? Quali sono i dati da produrre?

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 dato 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 totale.

**Fase 2** Risolvete 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,  $1/3 \times 10^2 \times 10 = 333.3 \text{ cm}^3$ , 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 totale 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  $125^{1/2}$ . 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 delle quattro facce triangolari della nostra piramide.

Calcolato tutto questo a mano, siamo ora pronti per descrivere i calcoli in Java.

**Fase 3** Progettate una classe che svolga i vostri 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 quindi 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** Scrivete 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** Realizzate la classe

In conseguenza della decisione presa nella Fase 3, abbiamo, come variabili di esemplare, 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 e il suo codice si trova nella cartella ch04/pyramid del pacchetto dei file scaricabili per questo libro.

#### Fase 6 Collaudate la vostra 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.

## 4.5 Stringhe

Molti programmi elaborano testi, composti da caratteri: lettere, numeri, segni di punteggiatura, spazi e così via. Una stringa è una sequenza di caratteri, come "Hello, World!", e in questo paragrafo vedrete come si elaborano stringhe in Java.

### 4.5.1 La classe String

In Java, le stringhe sono oggetti appartenenti alla classe **String** (si capisce che è il nome di una classe perché inizia con una lettera maiuscola, mentre i tipi di dati primitivi, come **int** e **double**, iniziano con una minuscola).

Per creare un oggetto di tipo stringa non c'è bisogno di invocare un costruttore: racchiudendo semplicemente una sequenza di caratteri fra virgolette si ottiene un *letterale* di tipo stringa. Ad esempio, la stringa letterale "Harry" è un oggetto della classe **String**.

In una stringa, il numero di caratteri ne costituisce la *lunghezza*. Come avete visto nel Capitolo 2, potete calcolare la lunghezza di una stringa mediante il metodo **length**.

Una stringa è una sequenza di caratteri. Le stringhe sono oggetti della classe **String**.

Per esempio, "Hello".length() vale 5 e la lunghezza di "Hello, World!" è 13 (le virgolette non fanno parte della stringa e non contribuiscono alla sua lunghezza, mentre spazi e segni di punteggiatura vanno conteggiati).

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 mettere insieme stringhe per formare una stringa più lunga: un processo denominato *concatenazione*.

```
String name = "Dave";
String message = "Hello, " + name;
```

Usando l'operatore + si possono concatenare stringhe, cioè porne una di seguito a un'altra per formare una stringa più lunga.

L'operatore + concatena due stringhe, a patto che almeno una delle due espressioni, poste a destra e a sinistra dell'operatore +, sia una stringa: in tal caso, anche l'altra espressione viene automaticamente convertita in una stringa, per poi concatenarle.

Per esempio, esaminiamo questo codice:

```
String a = "Agent";
int n = 7;
String bond = a + n;
```

Poiché a è una stringa, il numero intero n viene convertito nella stringa "7". Poi, le due stringhe "Agent" e "7" vengono concatenate, dando luogo alla stringa "Agent7".

La concatenazione è molto utile per ridurre il numero degli enunciati System.out.print. Per esempio, potete combinare:

```
System.out.print("The total is ");
System.out.println(total);
```

nella singola invocazione:

```
System.out.println("The total is " + total);
```

Se uno degli argomenti dell'operatore + è una stringa, l'altro viene convertito in una stringa.

La concatenazione "The total is " + total restituisce una stringa unica, formata dalla stringa "The total is " seguita dalla stringa corrispondente al numero contenuto in total.

### 4.5.3 Convertire stringhe in numeri

A volte si ha una stringa che contiene un numero, solitamente un dato inserito dall'utente. Supponete, ad esempio, che la variabile input, di tipo stringa, abbia il valore "19". Per ottenere il valore intero 19, si usa il metodo statico parseInt della classe Integer.

```
int count = Integer.parseInt(input); // count contiene il numero 19
```

Se una stringa contiene le cifre di un numero, per ottenerne il valore numerico usate il metodo Integer.parseInt oppure Double.parseDouble.

Per convertire una stringa che contiene le cifre di un numero in virgola mobile, ottenendo quindi il corrispondente valore in virgola mobile, si usa il metodo statico parseDouble della classe Double. Ad esempio, supponiamo che input contenga la stringa "3.95".

```
double price = Double.parseDouble(input); // price contiene il numero 3.95
```

Se la stringa contiene degli spazi o altri caratteri che non possono essere presenti all'interno di numeri, invocando questi metodi si verifica un errore: per il momento faremo l'ipotesi che i dati inseriti dall'utente non contengano mai caratteri non consentiti.

#### 4.5.4 Sottostringhe

Per estrarre una porzione di una stringa, usate il metodo `substring`.

Il metodo `substring` genera una sottostringa di una stringa. L'invocazione:

```
s.substring(start, pastEnd)
```

restituisce una stringa costituita da caratteri consecutivi della stringa `s`, 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"
```

Le posizioni dei caratteri in una stringa si contano a partire da zero.

Questa operazione `substring` costruisce una stringa formata da cinque caratteri, copiati dalla stringa `greeting`. Un aspetto curioso della funzione `substring` è la numerazione delle posizioni iniziali e finali: la prima posizione della stringa è contrassegnata dall'indice 0, la seconda dall'indice 1 e così via. Per esempio, la Figura 3 mostra i numeri che indicano le posizioni dei caratteri nella stringa `greeting`.

Il numero che indica la posizione dell'ultimo carattere nella stringa (12 per "Hello, World!") corrisponde sempre alla lunghezza della stringa, diminuita di 1.

Vediamo come si estrae la sottostringa "World". Il conteggio dei caratteri inizia da 0, non da 1, e potete constatare che `W`, l'ottavo carattere, ha la posizione 7. Il primo carattere che *non si vuole* copiare, il punto esclamativo, è nella posizione 12 (si veda la Figura 4). Pertanto, l'enunciato corretto per l'estrazione della sottostringa desiderata è:

```
String sub2 = greeting.substring(7, 12);
```

È curioso che si debba specificare la posizione del primo carattere desiderato e, poi, quella del primo che non si vuole. Questa impostazione 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` omettete 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:

**Figura 3**

Posizioni in una stringa

H	e	l	l	o	,		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11	12

**Figura 4**  
Estrazione  
di una sottostringa

H	e	l	l	o	,		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11	12

5

```
String tail = greeting.substring(7); // copia dalla posizione 7 alla fine
tail contiene la stringa "World!".
```

Se indicate una posizione non valida (un numero negativo oppure un valore maggiore della lunghezza della stringa), il programma termina con un messaggio d'errore.

In questo paragrafo abbiamo ipotizzato che ciascun carattere presente nella stringa occupi una sola posizione: ipotesi che, sfortunatamente, non è sempre corretta. Se elaborate stringhe che contengono caratteri di alcuni alfabeti internazionali oppure simboli speciali, alcuni di essi possono occupare due posizioni, come potrete vedere in Argomenti avanzati 4.5.



## Auto-valutazione

14. Ipotizzando che la variabile `s` di tipo `String` contenga il valore "Agent", che effettua produce l'assegnamento `s = s + s.length()` ?
15. Ipotizzando che la variabile `river` di tipo `String` contenga il valore "Mississippi", che valore ha l'espressione `river.substring(1, 2)`? E l'espressione `river.substring(2, river.length() - 3)` ?



## Consigli per la produttività 4.1

### 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)
```

A questo punto, un numero sorprendente di studenti semplicemente abbandona l'impresa, dicendo "non funziona", oppure "il mio programma è morto", senza nemmeno leggere 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 linea della segnalazione e termina con `Exception`. Se viene segnalata una `StringIndexOutOfBoundsException`, allora c'è stato un problema relativamente all'accesso a una posizione non valida all'interno di una stringa: questa è un'utile informazione.

Il numero della riga del file sorgente errato 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 quando si è verificata

l'eccezione. La prima linea della traccia riguarda il metodo che ha effettivamente generato l'eccezione, mentre l'ultima linea 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 linea del *vostro codice* che appare nella segnalazione di eccezione. Ad esempio, saltate le linee che si riferiscono a

```
java.lang.String.substring(String.java:1444)
```

La linea 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

### Sequenze di escape

Immaginate di voler visualizzare una stringa che contiene virgolette, come la seguente:

```
Hello, "World"!
```

Non potete scrivere:

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

perché, non appena il compilatore legge "Hello, ", deduce che la stringa sia terminata e, poi, rimane completamente disorientato dalla parola `World` seguita da due coppie di virgolette. Un lettore umano probabilmente capirebbe che il secondo e il terzo paio di virgolette sono, nelle intenzioni di chi scrive, parti della stringa, ma i compilatori ragionano a senso unico e, quindi, se alla prima analisi l'enunciato non ha senso, si rifiutano semplicemente di proseguire e segnalano un errore. Come fare, allora, per visualizzare le virgolette sullo schermo? Basta far precedere le virgolette all'interno della stringa da un carattere *barra rovesciata* (*backslash*, \). All'interno di una stringa, la sequenza \" indica le virgolette in senso letterale, non la fine dei caratteri. Di conseguenza, l'enunciato corretto per la visualizzazione è:

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

Il carattere barra rovesciata viene qui usato per definire un cosiddetto "carattere di *escape*" (o di uscita, nel senso di "uscita dalla normale interpretazione dei caratteri") e la sequenza di caratteri \" si chiama "sequenza di *escape*". La barra rovesciata non rappresenta se stessa: si usa per codificare altri caratteri che, altrimenti, sarebbe difficile inserire in una stringa.

A questo punto, come vi comportate se volete stampare proprio una barra rovesciata (per esempio, per specificare il percorso di un file in Windows)? Bisogna inserirne due consecutive, in questo modo:

```
System.out.println("The secret message is in C:\\\\Temp\\\\Secret.txt");
```

Questo enunciato stamperà:

```
The secret message is in C:\Temp\Secret.txt
```

Un'altra sequenza di escape che talvolta si usa è `\n`, che indica una riga nuova (o il carattere di ritorno carrello in una stampante, *newline*): stampare un carattere *newline* produce l'inizio della nuova riga sullo schermo. Per esempio, l'enunciato:

```
System.out.print("*\n**\n***\n");
```

visualizza:

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

in tre righe separate. Naturalmente, si poteva ottenere lo stesso risultato anche con tre distinte invocazioni di `println`.

Infine, le sequenze di escape sono utili per inserire caratteri internazionali in una stringa. Per esempio, supponete di volere stampare “All the way to San José!”, con la lettera accentata “é”: se utilizzate una tastiera americana, manca il tasto per digitare questo carattere. Per rappresentare i caratteri internazionali, Java utilizza uno schema di codifica chiamato *Unicode*: per esempio, il carattere “é” corrisponde alla codifica `00E9`. Possiamo inserire questo carattere in una stringa, scrivendo `\u`, seguito dalla sua codifica Unicode:

```
System.out.println("All the way to San Jos\u00E9!");
```

Potete trovare i codici per migliaia di caratteri nel sito [www.unicode.org](http://www.unicode.org).



## Argomenti avanzati 4.5

### Le stringhe e il tipo `char`

Le stringhe sono sequenze di caratteri Unicode (si vedano le Note di cronaca 4.2). I letterali di tipo carattere sembrano stringhe letterali, ma sono delimitate da apici singoli anziché doppi: '`H`' è un carattere, '`"H"`' è una stringa che contiene un solo carattere.

I caratteri hanno valori numerici: ad esempio, il carattere '`H`' viene in realtà codificato con il numero 72.

Anche le sequenze di escape (viste in Argomenti avanzati 4.4) possono figurare all'interno di letterali di tipo carattere: ad esempio, '`\n`' è il carattere per andare a capo e '`\u00E9`' è il carattere “é”.

Quando venne inizialmente progettato il linguaggio Java, ciascun carattere Unicode era codificato con due byte e il tipo di dati `char` aveva proprio lo scopo di contenere il codice di un carattere Unicode. Tuttavia, nel 2003 la codifica Unicode è stata estesa a tal punto che per alcuni caratteri è sorta la necessità di una codifica mediante una coppia di valori di tipo `char`, per cui non è più possibile pensare a un valore di tipo `char` come a un carattere. In termini più precisi, un valore di tipo `char` è una *unità di codice* (“code

unit") all'interno della codifica UTF-16 di Unicode: tale codifica rappresenta con un unico valore di tipo `char` i caratteri più comuni, usando una coppia di valori di tipo `char` soltanto per i caratteri meno comuni o *supplementari*.

Il metodo `charAt` della classe `String` restituisce un'unità di codice presente in una stringa. Come il metodo `substring`, le posizioni all'interno della stringa vengono contate a partire da 0. Ad esempio, l'enunciato

```
String greeting = "Hello";
char ch = greeting.charAt(0);
```

memorizza il carattere 'H' nella variabile `ch`.

Usando variabili di tipo `char`, però, i vostri programmi possono avere problemi con stringhe che contengono caratteri internazionali o simboli speciali. Ad esempio, il singolo carattere  $\mathbb{Z}$  (il simbolo matematico che rappresenta l'insieme dei numeri interi) viene codificato con le due unità di codice '\uD835' e '\uDD6B'.

Se invocate `charAt(0)` sulla stringa contenente il singolo carattere  $\mathbb{Z}$  (cioè la stringa "\uD835\uDD6B"), otterrete soltanto la prima metà di tale carattere supplementare.

Dovreste, quindi, usare valori di tipo `char` soltanto quando siete assolutamente certi che non avrete bisogno di codificare caratteri supplementari.

## 4.6 Leggere dati in ingresso

I programmi Java che avete scritto finora hanno costruito oggetti, invocato metodi, visualizzato risultati, per poi terminare la propria esecuzione: non erano programmi interattivi e non ricevevano dati in ingresso dall'utente. In questo paragrafo imparerete un metodo per acquisire dati in ingresso.

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 programmatore avrebbero scritto applicazioni dotate di interfaccia grafica, con menu e campi di testo. All'oggetto `System.in` venne associato un insieme di funzionalità davvero minimale: è in grado di leggere soltanto un byte per volta. Finalmente, nella versione 5 di Java è stata aggiunta una classe `Scanner` che consente la lettura semplice e comoda di dati inseriti in ingresso tramite tastiera.

Per costruire un oggetto di tipo `Scanner`, fornite semplicemente al costruttore di `Scanner` l'oggetto `System.in`:

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

Potete creare un oggetto di questo tipo a partire da qualsiasi flusso di ingresso (come, ad esempio, un file), ma solitamente lo userete per leggere i dati da tastiera, ricevuti tramite `System.in`.

Dopo aver creato uno scanner ("scansionatore"), ne potete usare i metodi `nextInt` o `nextDouble` per leggere il successivo numero, rispettivamente intero o in virgola mobile:

```
System.out.print("Enter quantity: ");
```

Per ricevere in ingresso dati inseriti in una finestra di console usate la classe `Scanner`.

```

int quantity = in.nextInt();

System.out.print("Enter price: ");
double price = in.nextDouble();

```

Quando viene invocato il metodo `nextInt` o il metodo `nextDouble`, il programma si arresta in attesa che l'utente digiti un numero e prema sulla tastiera il tasto “Enter” (“Invio”) per questo motivo, prima di invocare un metodo di `Scanner` dovreste sempre fornire adeguate istruzioni all'utente (come, nell'esempio, “Enter quantity: ”). Un messaggio di questo tipo viene chiamato *prompt*.

Se l'utente inserisce un dato che non è un numero, si ha un'eccezione che interrompe l'esecuzione. Nel prossimo Capitolo vedrete come si possa verificare se l'utente ha correttamente introdotto un dato numerico.

Il metodo `nextLine` restituisce, sotto forma di oggetto di tipo `String`, la successiva riga di testo fornita in ingresso, fino alla pressione del tasto “Enter” da parte dell'utente. Il metodo `next` restituisce, invece, la *parola* successiva, definita come sequenza di caratteri terminata da un *carattere di spaziatura* (“white space”): uno spazio, un carattere di fine riga o un carattere di tabulazione.

```

System.out.print("Enter city: ");
String city = in.nextLine();

System.out.print("Enter state code: ");
String state = in.next();

```

In questo caso abbiamo usato il metodo `nextLine` per leggere il nome di una città, perché può essere composto da più parole, come `San Francisco`. Abbiamo, invece, usato il metodo `next` per leggere la sigla dello stato (come `CA`) perché è certamente composta da una sola parola.

Ecco, come esempio, un programma che riceve dati in ingresso: usa la classe `CashRegister` e simula una transazione in cui un utente acquista un articolo, lo paga e riceve il resto.

La classe viene chiamata `CashRegisterSimulator` e non `CashRegisterTester`, perché riserviamo il suffisso `Tester` alle classi che hanno l'unico scopo di collaudarne altre.

### File ch04/cashregister/CashRegisterSimulator.java

```

import java.util.Scanner;

/**
 * Programma che simula una transazione in cui
 * un utente paga un articolo e riceve il resto.
 */
public class CashRegisterSimulator
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);

        CashRegister register = new CashRegister();

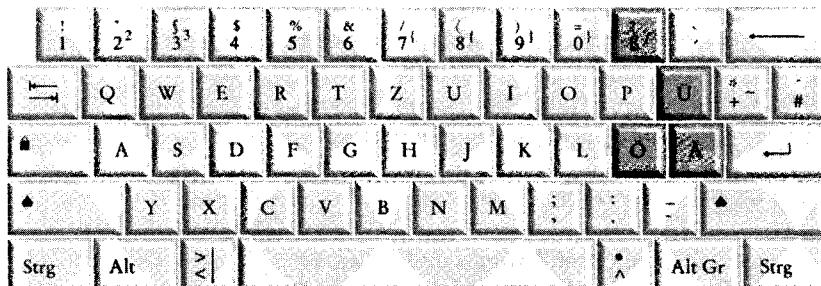
```



## Note di cronaca 4.2

### Alfabeti internazionali

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 che da sinistra a destra e molte di queste lingue hanno simboli che vanno posizionati sopra o sotto altri caratteri, come accade nella lingua thai (si notino nella figura i simboli accompagnati da un cerchietto). Ciascuno di questi alfabeti ha un numero di lettere compreso fra 30 e 100.

La situazione è ancora più complessa nelle lingue che usano la scrittura cinese: i dialetti cinesi, il giapponese e il coreano. La scrittura cinese non è alfabetica ma *ideografica*: un carattere rappresenta un concetto o una cosa e non un singolo suono. Sono attivamente utilizzati decine di migliaia di ideogrammi.

Le incoerenze fra le codifiche dei caratteri hanno rappresentato un grave problema per la comunicazione elettronica internazionale e per i produttori di software in competizione



Ideogrammi cinesi

chiamato *Unicode*, appositamente progettato per essere in grado di codificare un testo in tutte le lingue scritte del mondo ([www.unicode.org](http://www.unicode.org)). Nella prima versione di Unicode sono stati assegnati codici a circa 39 000 caratteri, compresi 21 000 ideogrammi cinesi, scegliendo un codice di due byte (in grado di codificare, quindi, circa 65 000 caratteri). Si presumeva che rimanesse spazio per codificare anche lingue strane, come i geroglifici egiziani e l'antica lingua scritta dell'isola di Java.

Java è stato uno dei primi linguaggi di programmazione ad adottare lo standard Unicode. Tutti i caratteri Unicode possono essere memorizzati in stringhe Java, ma quali possano essere di fatto visualizzati dipende dal sistema operativo del vostro computer. Il tipo di dato primitivo *char* viene utilizzato per memorizzare un carattere Unicode a due byte.

Sfortunatamente, nel 2003 accadde l'inevitabile: venne aggiunta allo standard Unicode una gran quantità di ideogrammi cinesi, superando il limite dei 16 bit. Oggi alcuni caratteri devono necessariamente essere codificati con una coppia di valori di tipo *char*, come detto in Argomenti avanzati 4.5.

କ	ଖ	ଗ	ଙ	ତ	ଶ	ପ	ହ
ବ	ବୁ	ମ	ମୁ	ରୁ	ରୁଣ	ଲୁ	ଲୁଣ
ଳ	ଳୁ	.	ଳୁମୁ	ମୁଣ୍ଡୁ	ମୁଣ୍ଡୁନ	ମୁଣ୍ଡୁନୁ	ମୁଣ୍ଡୁନୁଣ
ଛ	ଛୁ	ଚ	ଚୁ	ଚୁଣୁ	ଚୁଣୁନ	ଚୁଣୁନୁ	ଚୁଣୁନୁଣ
ଫ	ଫୁ	ଫ	ଫୁଣୁ	ଫୁଣୁନ	ଫୁଣୁନୁ	ଫୁଣୁନୁଣ	ଫୁଣୁନୁଣୁ
ବୀ	ବୀରୁ	ବୀ	ବୀରୁଣୁ	ବୀରୁଣୁନ	ବୀରୁଣୁନୁ	ବୀରୁଣୁନୁଣ	ବୀରୁଣୁନୁଣୁ
ପୀ	ପୀରୁ	ପୀ	ପୀରୁଣୁ	ପୀରୁଣୁନ	ପୀରୁଣୁନୁ	ପୀରୁଣୁନୁଣ	ପୀରୁଣୁନୁଣୁ
ମୀ	ମୀରୁ	ମୀ	ମୀରୁଣୁ	ମୀରୁଣୁନ	ମୀରୁଣୁନୁ	ମୀରୁଣୁନୁଣ	ମୀରୁଣୁନୁଣୁ

Una selezione dell'alfabeto thai

```

        System.out.print("Enter price: ");
        double price = in.nextDouble();
        register.recordPurchase(price);

        System.out.print("Enter dollars: ");
        int dollars = in.nextInt();
        System.out.print("Enter quarters: ");
        int quarters = in.nextInt();
        System.out.print("Enter dimes: ");
        int dimes = in.nextInt();
        System.out.print("Enter nickels: ");
        int nickels = in.nextInt();
        System.out.print("Enter pennies: ");
        int pennies = in.nextInt();
        register.enterPayment(dollars, quarters, dimes, nickels, pennies);

        System.out.print("Your change: ");
        System.out.println(register.giveChange());
    }
}

```

### Esecuzione del programma

```

Enter price: 7.55
Enter dollars: 10
Enter quarters: 2
Enter dimes: 1
Enter nickels: 0
Enter pennies: 0
Your change: 3.05

```



### Auto-valutazione

16. Perché non si possono leggere dati in ingresso direttamente tramite `System.in`?
17. Ipotizzate che `in` sia un oggetto di tipo `Scanner` che legga da `System.in` e che il vostro programma invochi

```
String name = in.next();
```

Cosa viene memorizzato in `name` se l'utente digita John Q. Public?



### Esempi completi 4.2

#### Estrazione delle iniziali

Avete il compito di scrivere un programma che chiede all'utente il suo nome e quello del suo partner, per poi comporre una stringa con le iniziali, unite da un carattere & (“ampersand”, in italiano “e commerciale”), come R&S.

Lo pseudocodice è veramente semplice:

Leggere il primo nome e memorizzarlo nella variabile stringa “first”.

Leggere il secondo nome e memorizzarlo nella variabile stringa “second”.

Comporre la stringa "initials" con il primo carattere di "first", il carattere & e il primo carattere di "second".  
Visualizzare la stringa "initials".

Per leggere uno dei nomi, lo chiediamo all'utente scrivendo un messaggio appropriato (*prompt*) e invocando, poi, un metodo di un oggetto di tipo **Scanner**:

```
Scanner in = new Scanner(System.in);
System.out.print("Enter your first name: ");
String first = in.____;
```

Per leggere una stringa si può invocare **next** oppure **nextLine**. Se sapete che i dati in ingresso sono costituiti da un'unica parola, senza spazi interposti, invocate **next**. Supponiamo, però, che vogliate leggere Mary Lou, con uno spazio tra le due parole. Se usate **next**, viene letto soltanto Mary e la stringa Lou verrà letta come secondo nome: non è proprio quello che volete, dato che Mary Lou vorrebbe che la prima lettera del proprio nome, M, venisse unita alla prima lettera del nome del suo partner, non a quella del proprio secondo nome. In questo caso, quindi, il metodo giusto da invocare è **nextLine**.

Ora, dobbiamo scoprire come estrarre da una stringa la sua prima lettera. L'invocazione **first.substring(0, 1)** restituisce una stringa composta da un unico carattere, copiato dalla posizione iniziale di **first**.

L'invocazione **second.substring(0, 1)** ottiene il medesimo risultato operando sul secondo nome. Infine, dobbiamo concatenare le corte stringhe così ottenute con la stringa "&", ottenendo una stringa di lunghezza 3 da memorizzare nella variabile **initials**.

```
String initials = first.substring(0, 1) + "&" + second.substring(0, 1);
```

Questa figura illustra l'intero procedimento:

first =	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>M</td><td>a</td><td>d</td><td>a</td><td>l</td><td>f</td><td>o</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table>	M	a	d	a	l	f	o	0	1	2	3	4	5	6
M	a	d	a	l	f	o									
0	1	2	3	4	5	6									
second =	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>S</td><td>a</td><td>l</td><td>l</td><td>y</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> </table>	S	a	l	l	y	0	1	2	3	4				
S	a	l	l	y											
0	1	2	3	4											
initials =	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>R</td><td>&amp;</td><td>S</td></tr> <tr><td>0</td><td>1</td><td>2</td></tr> </table>	R	&	S	0	1	2								
R	&	S													
0	1	2													

Ecco il programma completo:

### File ch04/initials/Initials.java

```
import java.util.Scanner;

public class Initials
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your first name: ");
```

```

        String first = in.nextLine();
        System.out.print("Enter your significant other's first name: ");
        String second = in.nextLine();
        String initials = first.substring(0, 1) + "&" + second.substring(0, 1);
        System.out.println(initials);
    }
}

```

## Esecuzione del programma

```

Enter your first name: Rodolfo
Enter your significant other's first name: Sally
R&S

```



## Argomenti avanzati 4.6

### Visualizzare numeri nel formato desiderato

Il formato standard per stampare numeri non è sempre quello più adatto. Per esempio, esaminate questo codice:

```

double total = 3.50;
final double TAX_RATE = 8.5; // aliquota d'imposta in percentuale
double tax = total * TAX_RATE / 100; // l'imposta è 0.2975
System.out.println("Total: " + total);
System.out.println("Tax: " + tax);

```

La sua esecuzione visualizza:

```

Total: 3.5
Tax: 0.2975

```

Preferiremmo visualizzare gli importi con due cifre dopo il separatore decimale, in questo modo:

```

Total: 3.50
Tax: 0.30

```

Ciò si può ottenere usando il metodo `printf` della classe `PrintStream` (di cui, come sapete, `System.out` è un esemplare). Il primo parametro del metodo `printf` è una *stringa di formato* che indica il modo in cui vanno visualizzati i dati: contiene caratteri da stampare e caratteri che fungono da *indicatori di formato*, cioè codici che iniziano con un carattere % e terminano con una lettera che indica il tipo di formato. Esiste un certo numero di formati, i più importanti dei quali sono riassunti nella Tabella 4.1 restanti parametri di `printf` sono i valori da visualizzare secondo il formato richiesto. Ad esempio:

```
System.out.printf("Total:%5.2f", total);
```

visualizza la stringa `Total:` seguita da un numero in virgola mobile *lungo* 5 cifre con una *precisione* di 2 cifre. La lunghezza del numero è il numero totale di caratteri da visualizzare: nel nostro caso, uno spazio, la cifra 3, un punto e due cifre. Se aumentate questa

lunghezza verranno aggiunti ulteriori spazi iniziali. La precisione, invece, è il numero di cifre presenti dopo il separatore decimale.

**Tabella 4**  
Tipi di formato

Codice	Tipo	Esempio
d	Intero decimale	123
x	Intero esadecimale	7B
o	Intero ottale	173
f	Virgola mobile	12.30
e	Virgola mobile esponenziale	1.23e+1
g	Virgola mobile generico (notazione esponenziale per i numeri molto grandi o molto piccoli)	12.3
s	Stringa	Tax:
n	Fine riga indipendente dalla piattaforma	

**Tabella 5**  
Modificatori di formato  
(flag)

Codice	Significato	Esempio
-	Allinea a sinistra	1.23 seguito da spazi
0	Mostra gli zeri iniziali	001.23
+	Mostra il segno più per numeri positivi	+1.23
(	Racchiude tra parentesi i numeri negativi	(1.23)
,	Mostra il separatore di migliaia	12,300
^	Usa lettere maiuscole	12.3E+1

Questo utilizzo semplificato di `printf` è sufficiente nella maggior parte dei casi, ma di tanto in tanto vedrete esempi più complessi, come questo:

```
System.out.printf("%-6s%5.2f%n", "Tax:", total);
```

In questo caso vediamo tre indicatori di formato. Il primo è `%-6s`: s indica una stringa, il segno meno è un *flag* che modifica il formato e, in particolare, indica allineamento a sinistra (nella Tabella 5 potete vedere i più comuni modificatori di formato, che devono essere inseriti subito dopo il carattere %). Se la stringa da visualizzare è più corta della lunghezza indicata, viene inserita a sinistra e vengono aggiunti a destra gli spazi necessari (in mancanza di modificatore di formato, i dati vengono invece allineati a destra, inserendo spazi a sinistra). Di conseguenza, `%-6s` indica una stringa di lunghezza 6 allineata a sinistra.

Il formato `%5.2f` è già stato esaminato: indica un numero in virgola mobile di lunghezza 5 e precisione 2. L'indicatore finale, `%n`, specifica di andare a capo in un formato indipendente dalla piattaforma. In Windows le righe vanno terminate con *due* caratteri: un “ritorno carrello”, ‘\r’, e un carattere di “nuova riga”, ‘\n’. In altri sistemi operativi è sufficiente il solo carattere ‘\n’. Il formato `%n` produce i terminatori di riga appropriati.

Oltre alla stringa di formato, questa invocazione di `printf` ha altri due parametri; al metodo `printf` può essere fornito un numero qualsiasi di parametri, che, ovviamente, devono trovare corrispondenza negli indicatori presenti nella stringa di formato.

Il metodo `format` della classe `String` è simile al metodo `printf`, ma, anziché visualizzare dati, restituisce una stringa. Ad esempio, l'invocazione

```
String message = String.format("Total:%.2f", total);
```

assegna alla variabile `message` la stringa "Total: 3.50".

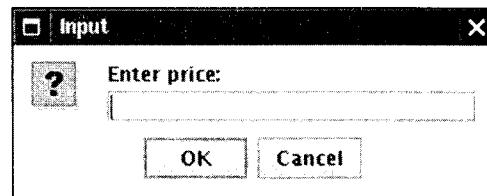


## Argomenti avanzati 4.7

### Ricezione 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 ricevere dati



Invocate il metodo statico `showInputDialog` della classe `JOptionPane`, fornendo come parametro la stringa che chiede il dato in ingresso all'utente. Ad esempio:

```
String input = JOptionPane.showInputDialog("Enter price:");
```

Tale metodo restituisce un oggetto di tipo `String`. 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);
```

Inoltre, ogniqualvolta invocate i metodi `showInputDialog` o `showMessageDialog` nei vostri programmi, dovete aggiungere alla fine del metodo `main` la riga seguente:

```
System.exit(0);
```

I metodi `showInputDialog` e `showMessageDialog` provocano l'avvio di un *thread* (flusso di esecuzione) relativo all'interfaccia utente per la gestione dei dati in ingresso. Quando il metodo `main` termina la propria esecuzione, quel thread è ancora in esecuzione e il programma non termina in modo automatico: per provocare la terminazione forzata di un programma è necessario invocare il metodo `exit` della classe `System`. Il parametro del metodo `exit` è il *codice di stato* del programma: il codice 0 indica la terminazione corretta, mentre si possono usare codici di stato diversi da zero per indicare varie condizioni di errore.

## 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 interi e due tipi in virgola mobile.
- Un calcolo numerico trabocca se il risultato esce dall'intervallo del tipo numerico.
- Avvengono errori di arrotondamento quando non è possibile fare una conversione numerica esatta.

### Uso delle costanti per chiarire il significato dei valori numerici

- Una variabile `final` è una costante: dopo che le è stato assegnato un valore, non può più essere modificata.
- Date un nome alle costanti, per rendere i vostri programmi più facili da leggere e da modificare.

### Espresioni aritmetiche in Java

- Gli operatori `++` e `--` aumentano e diminuiscono 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.
- La classe `Math` contiene i metodi `sqrt` e `pow` per calcolare radici quadrate e potenze.
- Per convertire un valore in un tipo diverso si usa il cast (*nomeTipo*).
- Per arrotondare un numero in virgola mobile all'intero più vicino usate il metodo `Math.round`.

### Metodi statici e metodi di esemplare

- Un metodo statico non agisce su un oggetto.

### Elaborazione di stringhe in Java

- Una stringa è una sequenza di caratteri. Le stringhe sono oggetti della classe `String`.
- Usando l'operatore `+` si possono concatenare stringhe, cioè porne una di seguito a un'altra per formare una stringa più lunga.
- Se uno degli argomenti dell'operatore `+` è una stringa, l'altro viene convertito in una stringa.
- Se una stringa contiene le cifre di un numero, per ottenerne il valore numerico usate il metodo `Integer.parseInt` oppure `Double.parseDouble`.
- Per estrarre una porzione di una stringa, usate il metodo `substring`.
- Le posizioni dei caratteri in una stringa si contano a partire da zero.

### Programmi che ricevono dati in ingresso

- Per ricevere in ingresso dati inseriti in una finestra di console usate la classe `Scanner`.

## Classi, oggetti e metodi presentati nel capitolo

<code>java.io.PrintStream</code>	<code>parseInt</code>
<code>printf</code>	<code>toString</code>
<code>java.lang.Double</code>	<code>java.lang.Math</code>
<code>parseDouble</code>	<code>E</code>
<code>java.lang.Integer</code>	<code>PI</code>
<code>MAX_VALUE</code>	<code>abs</code>
<code>MIN_VALUE</code>	<code>acos</code>

asin	substring
atan	java.lang.System
atan2	in
ceil	java.math.BigDecimal
cos	add
exp	multiply
floor	subtract
log	java.math.BigInteger
log10	add
max	multiply
min	subtract
pow	java.util.Scanner
round	next
sin	nextDouble
sqrt	nextInt
tan	nextLine
toDegrees	javax.swing.JOptionPane
toRadians	showInputDialog
java.lang.String	showMessageDialog
format	

## Esercizi di ripasso

- \*\* **Esercizio R4.1.** Scrivete in Java le seguenti espressioni matematiche.

$$s = s_0 + v_0 t + \frac{1}{2} g t^2$$

$$G = 4\pi^2 \frac{a^3}{P^2(m_1 + m_2)}$$

$$FV = PV \cdot \left(1 + \frac{INT}{100}\right)^{YRS}$$

$$c = \sqrt{a^2 + b^2 - 2ab \cos \gamma}$$

- \*\* **Esercizio R4.2.** Scrivete in notazione matematica le seguenti espressioni Java.

- a.  $dm = m * (\text{Math.sqrt}(1 + v / c) / (\text{Math.sqrt}(1 - v / c) - 1));$
- b.  $volume = \text{Math.PI} * r * r * h;$
- c.  $volume = 4 * \text{Math.PI} * \text{Math.pow}(r, 3) / 3;$
- d.  $p = \text{Math.atan2}(z, \text{Math.sqrt}(x * x + y * y));$

- \*\*\* **Esercizio R4.3.** Che cosa c'è di sbagliato in questa versione della formula dell'equazione di secondo grado?

$$\begin{aligned} x1 &= (-b - \text{Math.sqrt}(b * b - 4 * a * c)) / 2 * a; \\ x2 &= (-b + \text{Math.sqrt}(b * b - 4 * a * c)) / 2 * a; \end{aligned}$$

- \*\* **Esercizio R4.4.** Fornite un esempio di trabocco con numeri interi. Lo stesso esempio funzionerebbe correttamente se si usassero numeri in virgola mobile?

\*\* **Esercizio R4.5.** Fornite un esempio di un errore di arrotondamento con numeri in virgola mobile. Lo stesso esempio funzionerebbe correttamente se si usassero numeri interi e si passasse a un'unità di misura inferiore, per esempio centesimi invece di dollari, in modo che i valori non abbiano parte frazionaria?

\*\* **Esercizio R4.6.** Esaminate il codice seguente:

```
CashRegister register = new CashRegister();
register.recordPurchase(19.93);
register.enterPayment(20, 0, 0, 0, 0);
System.out.print("Change: ");
System.out.println(register.giveChange());
```

Il programma visualizza il totale come **0.0700000000000028**: spiegate perché. Date consigli per migliorare il programma in modo che gli utenti non rimangano perplessi.

\* **Esercizio R4.7.** Se **n** è un numero intero e **x** è un numero in virgola mobile, spiegate la differenza fra i due enunciati seguenti:

**n** = (int) **x**;

e

**n** = (int) Math.round(**x**);

\*\*\* **Esercizio R4.8.** Se **n** è un numero intero e **x** è un numero in virgola mobile, spiegate la differenza fra i due enunciati seguenti:

**n** = (int) (**x** + 0.5);

e

**n** = (int) Math.round(**x**);

Per quali valori di **x** gli enunciati forniscono lo stesso risultato? Per quali valori di **x** forniscono risultati diversi?

\* **Esercizio R4.9.** Prendete in esame il distributore automatico di francobolli realizzato nei Consigli pratici 4.1. Cosa succede se il metodo **givePennyStamps** viene invocato prima del metodo **giveFirstClassStamps**?

\* **Esercizio R4.10.** Spiegate la differenza fra **2**, **2.0**, **'2'**, **"2"** e **"2.0"**.

\* **Esercizio R4.11.** Spiegate quali operazioni esegue ciascuno di questi due frammenti di programma:

```
int x = 2;
int y = x + x;
```

e

```
String s = "2";
String t = s + s;
```

\*\* **Esercizio R4.12.** Se **x** è di tipo **int** e **s** è di tipo **String**, queste affermazioni sono vere o false?

- Integer.parseInt("") + x** ha lo stesso valore di **x**

- b. `"" + Integer.parseInt(s)` ha lo stesso valore di `s`
- c. `s.substring(0, s.length())` ha lo stesso valore di `s`

**★★ Esercizio R4.13.** Come si ottiene il primo carattere di una stringa? E l'ultimo? Come si elimina da una stringa il suo primo carattere? E l'ultimo?

**★★★ Esercizio R4.14.** Come si ottiene l'ultima cifra di un numero intero? E la prima? In pratica, se `n` contiene `23456`, come si verifica che la prima cifra sia `2` e che l'ultima sia `6`? Non convertite il numero in stringa. *Suggerimento:* `%, Math.log`.

**★★ Esercizio R4.15.** Questo capitolo contiene numerose raccomandazioni su variabili e costanti: per rendere il programma più facile da leggere e da gestire nel tempo. Riepilogate brevemente queste raccomandazioni.

**★★★ Esercizio R4.16.** Che cos'è una variabile `final`? Potete dichiarare una variabile `final` senza assegnarle contemporaneamente un valore? (Provate.)

**★ Esercizio R4.17.** Quali sono i risultati delle seguenti espressioni? Per ciascuna riga, assumete 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. `Math.sqrt(Math.sqrt(n))`
- e. `(int) Math.round(x)`
- f. `(int) Math.round(x) + (int) Math.round(y)`
- g. `1 - (1 - (1 - (1 - (1 - n))))`

**★ Esercizio R4.18.** Quali sono i risultati delle seguenti espressioni? Per ciascuna riga, assumete questi valori:

```
int n = 4;
String s = "Hello";
String t = "World";
```

- a. `s + t`
- b. `s + n`
- c. `n + t`
- d. `s.substring(1, n)`
- e. `s.length() + t.length()`

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

# 5

## Decisioni

### Obiettivi del capitolo

- Saper realizzare decisioni usando enunciati `if`
- Raggruppare enunciati in blocchi in modo efficace
- Imparare a confrontare numeri interi, numeri in virgola mobile, stringhe e oggetti
- Definire il corretto ordine delle decisioni nelle ramificazioni multiple e annidate
- Programmare condizioni usando variabili e operatori booleani
- Saper progettare collaudi che verifichino tutte le parti di un programma

I programmi che abbiamo visto finora erano in grado di eseguire calcoli velocemente e di rappresentare grafica, ma erano veramente poco flessibili. Eccetto che per le possibili variazioni dei dati in ingresso, a ogni loro esecuzione si comportavano sempre nello stesso modo. In programmi informatici non banali, una delle caratteristiche fondamentali è la capacità di prendere decisioni e di eseguire azioni diverse, in relazione alla natura dei dati in ingresso. L'obiettivo di questo capitolo è di imparare come programmare decisioni semplici e complesse.

## 5.1 L'enunciato if

I programmi per calcolatori hanno spesso bisogno di prendere *decisioni*, cioè intraprendere azioni diverse in dipendenza dal verificarsi di una determinata condizione.

Esaminiamo la classe presentata nel Capitolo 3 per rappresentare un conto bancario. Il metodo `withdraw` permette di prelevare dal conto tutto il denaro che si vuole: il saldo può ridursi fino a diventare negativo. Questo non è un modello realistico per un conto bancario, quindi realizzeremo diversamente il metodo `withdraw` in modo che non si possa prelevare un importo superiore al denaro che è presente nel conto. Questo significa che il metodo `withdraw` deve prendere una *decisione*, ovvero deve determinare se consentire il prelievo oppure no.

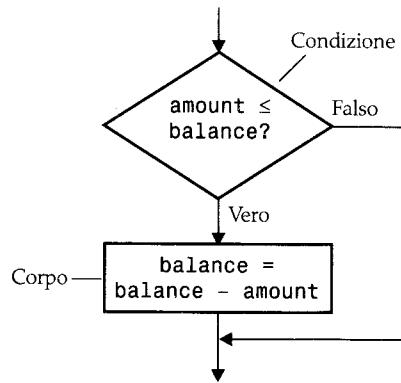
L'enunciato `if` consente a un programma di compiere azioni diverse in seguito alla verifica di una condizione.

Per realizzare una decisione si usa l'enunciato `if`, che è composto da due parti: una *condizione* e un *corpo*. Se la condizione è vera, viene eseguito il corpo dell'enunciato `if`, costituito, a sua volta, da un enunciato:

```
if (amount <= balance) // condizione
    balance = balance - amount; // corpo
```

L'enunciato di assegnazione verrà eseguito solo se l'importo da prelevare è minore o uguale al saldo, come rappresentato nella Figura 1.

**Figura 1**  
Diagramma di flusso di un enunciato if



Proviamo a rendere ancora più realistico il metodo `withdraw` della classe `BankAccount`. Molte banche non solo non permettono prelievi che superino il saldo del conto, ma addebitano anche una penale per ciascun tentativo.

Non potete semplicemente utilizzare due enunciati `if` complementari, in questo modo:

```
if (amount <= balance)
    balance = balance - amount;
if (amount > balance) // Usate, invece, if/else
    balance = balance - OVERDRAFT_PENALTY;
```

perché questa soluzione presenta due problemi. Se, per qualche motivo, modificherete l'enunciato `amount <= balance`, dovrete ricordarvi di aggiornare anche la condizione

`amount > balance`: non facendolo, la funzionalità del programma non sarebbe più corretta. Ancora più importante, se nel corpo del primo enunciato `if` modificate il valore di `balance` (come accade in questo esempio), la seconda condizione usa il nuovo valore.

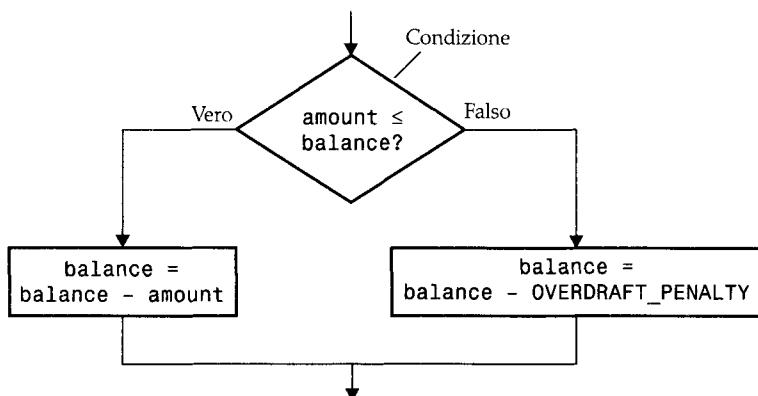
Per realizzare una scelta fra due alternative, dovete usare l'enunciato `if/else`:

```
if (amount <= balance)
    balance = balance - amount;
else
    balance = balance - OVERDRAFT_PENALTY;
```

Ora esiste una condizione sola: se viene soddisfatta, verrà eseguito il primo enunciato, altrimenti verrà eseguito il secondo. Il diagramma di flusso visibile nella Figura 2 fornisce una rappresentazione grafica del comportamento delle diramazioni.

**Figura 2**

Diagramma di flusso di un enunciato `if/else`



Un blocco di enunciati raggruppa più enunciati.

Abbastanza spesso il corpo dell'enunciato `if` è costituito da più enunciati, da eseguire in sequenza se la condizione è vera: tali enunciati vanno raggruppati insieme, per formare un *blocco di enunciati*, racchiudendoli fra parentesi graffe. Ecco un esempio:

```
if (amount <= balance)
{
    double newBalance = balance - amount;
    balance = newBalance;
}
```

In generale, il corpo di un enunciato `if` deve essere un blocco di enunciati, oppure un *enunciato semplice*, come questo:

```
balance = balance - amount;
```

oppure, ancora, un *enunciato composto* (ad esempio, un altro enunciato `if` oppure, come si vedrà nel Capitolo 6, un ciclo). Anche il corpo di un'alternativa `else` deve essere un enunciato, vale a dire un *enunciato semplice*, un *enunciato composto* o un blocco di enunciati.

## Sintassi di Java

### 5.1 L'enunciato if

#### Sintassi

```
if (condizione)
    enunciato
```

```
if (condizione)
    enunciato1
else
    enunciato2
```

#### Esempio

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.

Una condizione, che può essere vera o falsa. Spesso, come vedremo, si usano operatori relazionali: == != < <= > >=.

```
if (amount <= balance)
{
    balance = balance - amount;
}
else
{
    System.out.println("Insufficient funds");
    balance = balance - OVERDRAFT_PENALTY;
}
```

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.



#### Auto-valutazione

- Perché nell'esempio dedicato all'enunciato if/else abbiamo usato la condizione `amount <= balance` e non `amount < balance`?
- Qual è l'errore logico presente in questo codice? Come lo si può correggere?

```
if (amount <= balance)
    newBalance = balance - amount; balance = newBalance;
```



#### Consigli per la qualità 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: *Incolonnare* le parentesi graffe aperte e chiuse.

```
if (amount <= balance)
{
    double newBalance = balance - amount;
    balance = newBalance;
}
```

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 (amount <= balance) {
    double newBalance = balance - amount;
    balance = newBalance;
}
```

È una soluzione che fa risparmiare una linea di codice, ma è difficile individuare le parentesi graffe corrispondenti.

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.



## Consigli per la produttività 5.1

### Spostamenti verso destra e tabulazioni nel codice

Quando scrivete programmi Java, per *indicare i livelli di annidamento* nel codice si usano i *rientri*, cioè gli spostamenti verso destra:

```
public class BankAccount
{
    ...
    public void withdraw(double amount)
    {
        if (amount <= balance)
        {
            double newBalance = balance - amount;
            balance = newBalance;
        }
        ...
    }
    0 1 2 3
    Livelli di rientro
```

Quanti spazi bisogna usare per ciascun livello di rientro? Alcuni programmatori inseriscono otto spazi, ma non è una buona soluzione:

```
public class BankAccount
{
    ...
    public void withdraw(double amount)
    {
        if (amount <= balance)
        {
            double newBalance =
                balance - amount;
            balance = newBalance;
        }
    }
}
```

```

    }
    ...
}
```

Questa spaziatura addensa troppo il codice nella parte destra dello schermo e, di conseguenza, spesso bisogna spezzare le espressioni lunghe per riportarle su più righe. Per i rientri, due, tre o quattro spazi sono valori molto più comuni.

Per spostare il cursore dalla colonna più a sinistra al livello di rientro appropriato si può ovviamente premere la barra spaziatrice per il numero di volte sufficiente, ma molti programmati preferiscono usare il tasto di tabulazione (Tab), che sposta il cursore al successivo arresto di tabulazione. Per impostazione predefinita, gli arresti di tabulazione sono collocati ogni otto colonne, ma molti programmi per la scrittura di testi (*editor*) permettono di modificare questo valore. Di conseguenza, dovete scoprire come impostare gli arresti di tabulazione del vostro editor, affinché cadano, per esempio, ogni tre colonne.

Alcuni editor vi aiutano concretamente mediante una funzionalità di *rientro automatico* (*autoindent*), che inserisce automaticamente tutte le tabulazioni o gli spazi già presenti nella riga precedente, perché è abbastanza probabile che la nuova riga appartenga allo stesso livello di rientro. In caso contrario, dovete inserire o eliminare una tabulazione, ma questo rimane, comunque, un sistema più veloce rispetto all'inserimento manuale di tutte le tabulazioni a partire dal margine sinistro.

Se le tabulazioni sono comode per la digitazione del codice, alcuni editor le usano anche per allineare bene il testo: una soluzione non particolarmente efficiente, da momento che non esiste uno standard per l'ampiezza di un carattere di tabulazione e, addirittura, alcuni software li ignorano completamente. Ciò 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.



## Errori comuni 5.1

### Un punto e virgola dopo la condizione dell'`if`

Questo frammento di codice ha un errore davvero subdolo:

```

if (input < 0) ; // ERRORE
System.out.println("Bad input");
```

Dopo la condizione dell'`if` non ci dovrebbe essere il punto e virgola. Il compilatore interpreta quell'enunciato in questo modo: se `input` è minore di zero, esegui l'enunciato rappresentato dal solo punto e virgola, cioè l'enunciato “nullo”, che non fa niente. L'enunciato che segue il punto e virgola non fa più parte dell'`if` e viene quindi, sempre eseguito, visualizzando il messaggio d'errore con qualsiasi dato fornito in ingresso.



## Argomenti avanzati 5.1

### L'operatore di selezione o condizionale

Java prevede un operatore di selezione (o 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. Per esempio, possiamo calcolare il valore assoluto in questo modo:

$y = x \geq 0 ? x : -x;$

L'espressione precedente è una comoda scorciatoia, equivalente a questo frammento di codice:

```
if (x >= 0) y = x; else y = -x;
```

L'operatore di selezione è simile all'enunciato `if/else`, ma agisce a un livello sintattico diverso: elabora *valori* e restituisce un altro valore, mentre l'enunciato `if/else` raggruppa enunciati, dando luogo a un enunciato.

Sarebbe, quindi, un errore scrivere:

```
y = if (x > 0) x: else -x: // Error
```

Il costrutto sintattico **if/else** è un enunciato, non un valore, per cui non lo si può assegnare a una variabile.

In questo libro non useremo l'operatore di selezione, sebbene si tratti di un costrutto pratico e ammesso, che troverete in molti programmi Java.

## 5.2 Confrontare valori

### 5.2.1 Operatori relazionali

Un operatore relazionale verifica la relazione esistente tra due valori, come l'operatore `<=` che abbiamo già usato:

```
if (amount <= balance)
```

In Java esistono sei operatori relazionali:

Operatore Java	Notazione matematica	Descrizione
>	>	Maggiore
>=	$\geq$	Maggiore o uguale
<	<	Minore
<=	$\leq$	Minore o uguale
==	=	Uguale
!=	$\neq$	Diverso

Come potete vedere, soltanto due operatori relazionali di Java (< 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  $\geq$ ,  $\leq$  e  $\neq$  si ricordano facilmente, perché sono molto simili ai corrispondenti simboli matematici.

L'operatore `==` rappresenta la verifica di uguaglianza:

```
a = 5; // assegna 5 ad a
if (a == 5) ... // verifica se a è uguale a 5
```

Quindi, dovete ricordarvi l'utilizzo di `==` per verificare l'uguaglianza e l'uso di `=` per l'assegnazione.

Gli operatori relazionali hanno minore priorità rispetto agli operatori aritmetici: ciò significa che potete scrivere espressioni aritmetiche a sinistra e a destra di un operatore relazionale senza dover usare parentesi. Ad esempio, nell'espressione

```
amount + fee <= balance
```

vengono valutati i due operandi ai lati dell'operatore `<=`, `amount + fee` e `balance`, 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 confrontate numeri in virgola mobile dovete 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 stamperà:

```
sqrt(2) squared minus 2 is not 0 but 4.440892098500626E-16
```

Purtroppo, questi errori di arrotondamento sono inevitabili. Chiaramente, 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.

$$|x - y| \leq \epsilon$$

**Confrontando numeri in virgola mobile, non fate verifiche di uguaglianza, ma controllate se i valori sono sufficientemente prossimi.**

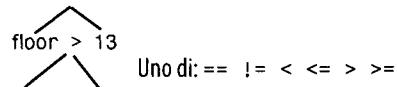
## Sintassi di Java

### 5.2 Confronti

#### Esempio

Queste quantità vengono confrontate.

Controllare se  
si è usata  
la direzione giusta:  
maggiore o minore?



Usare ==, non =.



```
String input;
if (input.equals("Y"))
```

Usare equals per confrontare stringhe.

```
double x; double y; final double EPSILON = 1E-14;
if (Math.abs(x - y) < EPSILON)
```

Verifica se questi numeri in virgola mobile sono sufficientemente prossimi.

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 un metodo chiamato `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
```

In realtà, questo enunciato verifica se le due variabili si riferiscono al *medesimo* oggetto di tipo stringa. Dal momento che possono esistere stringhe con contenuto identico conservate in oggetti diversi, questa verifica non ha mai senso nella programmazione reale: consultate Errori comuni 5.2.

Java distingue fra lettere maiuscole e minuscole: "Harry" e "HARRY" non sono stringhe uguali. Per fare confronti ignorando le differenze tra maiuscolo e minuscolo, usate il metodo `equalsIgnoreCase`:

```
if (string1.equalsIgnoreCase(string2)) ...
```

Il metodo `compareTo` confronta due stringhe secondo l'ordine alfabetico.

Se due stringhe non sono identiche, potreste voler conoscere la loro relazione reciproca: il metodo `compareTo` confronta le stringhe secondo l'ordine alfabetico. Se questa condizione è vera

```
string1.compareTo(string2) < 0
```

significa che `string1` precede la stringa `string2` nell'ordine alfabetico. Ciò succede, ad esempio, se `string1` contiene "Harry" e `string2` contiene "Hello". Se, invece, si verifica che

```
string1.compareTo(string2) > 0
```

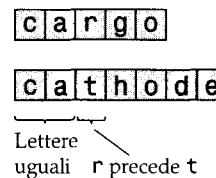
allora `string1` segue la stringa `string2` nell'ordine alfabetico. Infine, `string1` e `string2` sono uguali se

```
string1.compareTo(string2) == 0
```

In realtà, l'ordinamento "alfabetico" utilizzato in Java è leggermente diverso da quello di un normale dizionario. Innanzitutto, Java distingue fra maiuscolo e minuscolo; secondariamente, ordina i caratteri inserendo all'inizio i numeri, poi le lettere maiuscole e, infine, quelle minuscole. Per esempio, 1 precede B, che a sua volta precede a. Il carattere "spazio" precede tutti gli altri caratteri.

Esaminiamo da vicino il procedimento di confronto. Quando Java confronta due stringhe, vengono confrontate le lettere in posizioni corrispondenti a partire da sinistra, finché una delle stringhe termina oppure finché si incontra la prima differenza. Se una delle stringhe termina, tale stringa più breve precederà l'altra, più lunga. Se si trova un carattere diverso dall'altro, li si confronta per determinare quale stringa seguirà l'altra nell'ordine alfabetico. Questo processo si chiama confronto *lessicografico*. Per esempio, confrontiamo "car" con "cargo". In questo caso, le prime tre lettere sono uguali, quindi raggiungiamo la fine della prima stringa. Di conseguenza, "car" precede "cargo" nell'ordine lessicografico. Ora, confrontiamo "cathode" con "cargo": le prime due lettere corrispondono; poiché, nella terza posizione, r precede t, la stringa "cathode" segue "cargo" nell'ordinamento lessicografico (osservate la Figura 3).

**Figura 3**  
Confronto lessicografico



## Errori comuni 5.2

### Utilizzare == per confrontare stringhe

In Java, scrivere `==` quando si dovrebbe scrivere `equals` è un errore estremamente comune e succede specialmente con le stringhe. Se scrivete



Espressione	Valore	Commento
<code>3 &lt;= 4</code>	<code>true</code>	3 è minore di 4; <code>&lt;=</code> fa la verifica “minore o uguale”.
<code>3 &lt;= 4</code>	<b>Errore</b>	L’operatore “minore o uguale” è <code>&lt;=</code> , non <code>=&lt;</code> .
<code>3 &gt; 4</code>	<code>false</code>	<code>&gt;</code> è l’opposto di <code>&lt;=</code> .
<code>4 &lt; 4</code>	<code>false</code>	L’operando sinistro deve essere strettamente minore dell’operando destro.
<code>4 &lt;= 4</code>	<code>true</code>	I due operandi sono uguali; <code>&lt;=</code> fa la verifica “minore o uguale”.
<code>3 == 5 - 2</code>	<code>true</code>	<code>==</code> fa la verifica di uguaglianza.
<code>3 != 5 - 1</code>	<code>true</code>	<code>!=</code> fa la verifica di diversità ed è vero che 3 è diverso da $5 - 1$ .
<code>3 = 6 / 2</code>	<b>Errore</b>	Per la verifica di uguaglianza si usa <code>==</code> .
<code>1.0 / 3.0 == 0.3333333333</code>	<code>false</code>	Anche se i valori sono assai prossimi tra loro, non sono esattamente uguali (Errori comuni 4.3).
<code>"10" &gt; 5</code>	<b>Errore</b>	Non si può confrontare una stringa con un numero.
<code>"Tomato".substring(0, 3).equals("Tom")</code>	<code>true</code>	Per verificare se due stringhe hanno il medesimo contenuto usate sempre il metodo <code>equals</code> .
<code>"Tomato".substring(0, 3) == ("Tom")</code>	<code>false</code>	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).
<code>"Tom".equalsIgnoreCase("Tom")</code>	<code>true</code>	Se, in un confronto, non volete che maiuscole e minuscole siano considerate diverse, usate il metodo <code>equalsIgnoreCase</code> .

**Tabella 1**

Esempi con operatori  
relazionali

```
if (nickname == "Rob")
```

la condizione sarà vera solo se la variabile `nickname` si riferisce esattamente allo stesso oggetto stringa utilizzato per la stringa costante `"Rob"`. Per motivi di efficienza, Java crea un solo oggetto stringa per ciascuna stringa costante. Pertanto, la verifica seguente avrà esito positivo:

```
String nickname = "Rob";
...
if (nickname == "Rob") // la condizione è vera
```

Tuttavia, se la stringa che contiene le lettere `Rob` fosse stata costruita in qualche altro modo, allora la verifica avrebbe esito negativo:

```
String name = "Robert";
String nickname = name.substring(0, 3);
...
if (nickname == "Rob") // la condizione è falsa
```

Si tratta di una situazione particolarmente frustrante: il codice errato talvolta funziona e altre volte no. Dal momento che gli oggetti stringa sono sempre costruiti dal compilatore, non vi interesserà sapere se due oggetti stringa sono condivisi, per cui dovete ricordarvi di non usare mai l’operatore `==` per confrontare stringhe: utilizzate sempre i metodi `equals` o `compareTo`.

## 5.2.4 Confrontare oggetti

Confrontando due riferimenti a oggetti mediante l'operatore `==`, controllerete se i riferimenti puntano allo stesso oggetto. Ecco un esempio:

```
Rectangle box1 = new Rectangle(5, 10, 20, 30);
Rectangle box2 = box1;
Rectangle box3 = new Rectangle(5, 10, 20, 30);
```

Questo confronto risulta vero:

```
box1 == box2
```

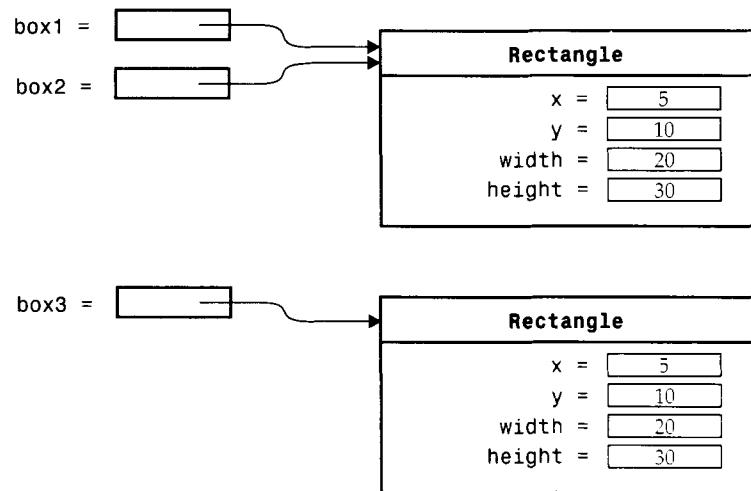
Infatti, entrambe le variabili si riferiscono allo stesso oggetto. Per contro, questo è falso:

```
box1 == box3
```

In questo caso, le variabili si riferiscono a oggetti *distinti* (osservate la Figura 4) e non importa che abbiano contenuti identici.

**Figura 4**

Confronto tra riferimenti



L'operatore `==` verifica se due riferimenti puntano allo stesso oggetto. Per confrontare, invece, i contenuti di oggetti, si deve usare il metodo `equals`.

Potete usare il metodo `equals` per verificare se due rettangoli hanno lo stesso *contenuto*, ovvero se hanno la stessa altezza, la stessa larghezza e l'angolo superiore sinistro nella stessa posizione. Per esempio, questa condizione è vera:

```
box1.equals(box3)
```

Usate, però, con attenzione il metodo `equals`: 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 10. Fino a quel momento, non potete usare il metodo `equals` per confrontare oggetti delle vostre classi.

## 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);
```

Noteate 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.

### Auto-valutazione

3. Qual è il valore di `s.length()` se `s` è:
  - a. la stringa vuota `""`?
  - b. la stringa `" "`, contenente soltanto uno spazio?
  - c. `null`?
4. Quali dei seguenti confronti contiene 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)`

### Consigli per la qualità 5.2

#### Evitare verifiche di condizioni che abbiano effetti collaterali

In Java, è ammesso annidare gli enunciati di assegnazione all'interno di quelli condizionali:

```
if ((d = b * b - 4 * a * c) >= 0) r = Math.sqrt(d);
```

È ammesso anche usare l'operatore di decremento all'interno di altre espressioni:

```
if (n-- > 0) ...
```

In ambedue i casi, si tratta di una pessima pratica di programmazione, perché si mescola una verifica con un'altra attività. L'altra attività (assegnare un valore alla variabile `d` o decrementare `n`) viene detta *effetto collaterale* della verifica.

Come vedremo in Argomenti avanzati 6.3, talvolta le condizioni con effetti collaterali possono essere d'aiuto per semplificare i cicli, ma dovrebbero essere sempre evitate negli enunciati `if`.



## 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.

Una librerie 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** Individuate la condizione per la diramazione.

Il nostro problema è talmente semplice che la condizione è, ovviamente, questa:

$$\text{prezzo originario} < 128?$$

Ci pare corretta e la useremo nella nostra soluzione.

Ovviamente, potete ottenere una soluzione altrettanto corretta scegliendo la condizione opposta: il prezzo originario è almeno pari a \$ 128? Probabilmente sceglierete questa seconda opzione se vi immedesimate nell'acquirente, che vuol sapere quando si applica lo sconto maggiore.

**Fase 2** Progettate lo pseudocodice per le operazioni da compiere quando la condizione è soddisfatta.

In questa fase elencate le azioni da intraprendere nel ramo “vero” del diagramma di flusso. I dettagli dipendono dal problema specifico: può darsi che vogliate visualizzare un messaggio, oppure calcolare qualche valore o, addirittura, terminare il programma.

Nel nostro esempio, dobbiamo applicare uno sconto dell'otto per cento:

$$\text{prezzo scontato} = 0.92 \times \text{prezzo originario}$$

**Fase 3** Progettate lo pseudocodice per le operazioni da compiere (se ce ne sono) quando la condizione *non* è soddisfatta.

Cosa volete fare se la condizione individuata nella Fase 1 non è soddisfatta? A volte non occorre far nulla: in tal caso usate un enunciato `if` privo di diramazione `else`.

Nel nostro esempio, la condizione ha verificato se il prezzo è inferiore a \$ 128: se ciò *non* è vero, il prezzo è almeno \$ 128, per cui si applica lo sconto del 16 per cento:

$$\text{prezzo scontato} = 0.84 \times \text{prezzo originario}$$

**Fase 4** Controllate con cura gli operatori relazionali.

Per prima cosa, assicuratevi che la verifica agisca nella giusta *direzione*: capita molto spesso di confondere < e >. Successivamente, valutate bene se occorra usare l'operatore < oppure il suo parente stretto, l'operatore <=.

Cosa bisogna fare se il prezzo è esattamente uguale a \$ 128? Leggendo con attenzione le specifiche del problema, vediamo che lo sconto inferiore viene applicato quando il prezzo originario è *minore* di \$ 128, mentre si applica lo sconto superiore quando il prezzo è *almeno* pari a \$ 128. Di conseguenza, un prezzo di \$ 128 *non* deve soddisfare la nostra condizione, per cui dobbiamo usare < e non <=.

**Fase 5** Eliminate le ripetizioni.

Verificate se vi siano azioni comuni a entrambe le diramazioni e spostatele all'esterno (come già suggerito in Consigli per la qualità 4.3).

Nel nostro esempio, abbiamo due enunciati aventi il medesimo formato:

$$\text{prezzo scontato} = \underline{\quad} \times \text{prezzo originario}$$

Differiscono soltanto per la percentuale di sconto. Nelle diramazioni è meglio impostare semplicemente tale percentuale, facendo i calcoli in seguito:

```
If prezzo originario < 128
    percentuale = 0.92
Else
    percentuale = 0.84
prezzo scontato = percentuale × prezzo originario
```

**Fase 6** Collaudate entrambe le diramazioni.

Progettate due situazioni diverse per il collaudo, una che soddisfi la condizione dell'enunciato *if* e una che non la soddisfi. Chiedetevi cosa dovrebbe succedere in ciascuno dei due casi e seguite, passo per passo, l'esecuzione dello pseudocodice.

Nel nostro esempio, consideriamo due diversi scenari per il prezzo originario: \$ 100 e \$ 200. Naturalmente, ci aspettiamo che lo sconto sia pari a 8 dollari nel primo caso e 32 dollari nel secondo.

Quando il prezzo originario è 100, la condizione  $100 < 128$  è vera e si ha

$$\begin{aligned}\text{percentuale} &= 0.92 \\ \text{prezzo scontato} &= 0.92 \times 100 = 92\end{aligned}$$

Quando il prezzo originario è 200, la condizione  $200 < 128$  è falsa e si ha

percentuale = 0.84  
 prezzo scontato =  $0.84 \times 200 = 168$

In entrambi i casi, quindi, otteniamo la risposta prevista.

**Fase 7** Scrivete l'enunciato `if` in Java.

Scrivete il codice in modo schematico, così

```
if ()  
{  
}  
else  
{  
}
```

poi riempitelo, seguendo le indicazioni viste in Sintassi di Java 5.1. Se non serve, evitate di scrivere la diramazione `else`.

L'enunciato completo relativo al nostro esempio è, quindi:

```
double HIGH_DISCOUNT_THRESHOLD = 128;  
double HIGH_RATE = 0.92;  
double LOW_RATE = 0.84;  
  
if (originalPrice < HIGH_DISCOUNT_THRESHOLD)  
{  
    rate = HIGH_RATE;  
}  
else  
{  
    rate = LOW_RATE;  
}  
discountedPrice = rate * originalPrice;
```

Abbiamo dato un nome ai valori costanti per facilitare la manutenzione del codice, come suggerito in Consigli per la qualità 4.1.



## Esempi completi 5.1

### Estrazione del carattere centrale

Abbiamo come obiettivo la creazione di una stringa contenente il carattere centrale estratto da una stringa assegnata, `str`. 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** Individuate la condizione per la diramazione.

Dobbiamo fare cose diverse per stringhe di lunghezza dispari e pari, per cui la condizione è

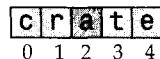
La lunghezza della stringa è dispari?

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** Progettate lo pseudocodice per le operazioni da compiere quando la condizione è soddisfatta.

Dobbiamo trovare la posizione del carattere centrale. Se la stringa ha lunghezza 5, la posizione è 2.



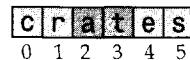
In generale:

```
posizione = str.length() / 2 (ignorando il resto)
risultato = str.substring(posizione, posizione + 1)
```

- Fase 3** Progettate lo pseudocodice per le operazioni da compiere (se ce ne sono) quando la condizione *non* è soddisfatta.

Anche in questo caso dobbiamo trovare le posizioni dei caratteri centrali. 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** Controllate 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** Eliminate le ripetizioni.

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 Collaudate 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 (ignorando il resto)
lunghezza = 2
risultato = str.substring(3, 5) = "it"

```

#### Fase 7 Scrivete 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);

```

## 5.3 Alternative multiple

### 5.3.1 Sequenze di confronti

Molte situazioni richiedono più di una singola decisione di tipo `if/else` e c'è bisogno di esprimere una sequenza di confronti tra loro correlati.

Il programma seguente chiede all'utente un valore che descriva la magnitudo di un terremoto, secondo la scala Richter, per poi stampare una descrizione del probabile impatto della scossa. La scala Richter è un sistema di misurazione della forza di un terremoto e ogni passaggio da un valore della scala al successivo, per esempio quello fra 6.0 e 7.0, indica una moltiplicazione per dieci della potenza del sisma. Il terremoto del 1989 a Loma Prieta, che danneggiò il Bay Bridge di San Francisco e che distrusse molti edifici in numerose città della baia, fece registrare 7.1 gradi della scala Richter.

#### File ch05/quake/Earthquake.java

```
/*
 * Una classe che descrive gli effetti di un terremoto.
 */
public class Earthquake
{
    private double richter;

    /**
     * Costruisce un oggetto che rappresenta un terremoto.
     * @param magnitude la magnitudo nella scala Richter
     */
    public Earthquake(double magnitude)
    {
        richter = magnitude;
    }

    /**
     * Restituisce una descrizione dell'effetto del terremoto.
     * @return la descrizione dell'effetto
     */
    public String getDescription()
    {
        String r;
        if (richter >= 8.0)
            r = "Most structures fall";
        else if (richter >= 7.0)
            r = "Many buildings destroyed";
        else if (richter >= 6.0)
            r = "Many buildings considerably damaged, some collapse";
        else if (richter >= 4.5)
            r = "Damage to poorly constructed buildings";
        else if (richter >= 3.5)
            r = "Felt by many people, no destruction";
        else if (richter >= 0)
            r = "Generally not felt by people";
        else
            r = "Negative numbers are not valid";
    }
}
```

Si possono combinare più condizioni  
per prendere decisioni articolate.  
La loro disposizione corretta dipende  
dalla logica del problema  
che si deve risolvere.

```

        return r;
    }
}
}
```

### File ch05/quake/EarthquakeRunner.java

```

import java.util.Scanner;

/**
     Programma che visualizza la descrizione di un terremoto
     con magnitudo assegnata.
 */
public class EarthquakeRunner
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);

        System.out.print("Enter a magnitude on the Richter scale: ");
        double magnitude = in.nextDouble();
        Earthquake quake = new Earthquake(magnitude);
        System.out.println(quake.getDescription());
    }
}
```

### Esecuzione del programma

```

Enter a magnitude on the Richter scale: 7.1
Many buildings destroyed
```

Qui dobbiamo ordinare le condizioni in modo che venga effettuato per primo il confronto con il valore di soglia maggiore. Supponiamo di invertire l'ordine delle verifiche:

```

if (richter >= 0) // Verifiche in ordine errato
    r = "Generally not felt by people";
else if (richter >= 3.5)
    r = "Felt by many people, no destruction";
else if (richter >= 4.5)
    r = "Damage to poorly constructed buildings";
else if (richter >= 6.0)
    r = "Many buildings considerably damaged, some collapse";
else if (richter >= 7.0)
    r = "Many buildings destroyed";
else if (richter >= 8.0)
    r = "Most structures fall";
```

In questo modo non funziona: tutti i valori non negativi di `richter` ricadono nel primo caso e le altre condizioni non verranno mai controllate.

In questo esempio è importante anche l'utilizzo della verifica mediante gli operatori `if/else/else`, invece di limitarsi a più enunciati `if` indipendenti. Osservate questa sequenza di verifiche indipendenti:

```

if (richter >= 8.0) // Non usa else
    r = "Most structures fall";
```

```

if (richter >= 7.0)
    r = "Many buildings destroyed";
if (richter >= 6.0)
    r = "Many buildings considerably damaged, some collapse";
if (richter >= 4.5)
    r = "Damage to poorly constructed buildings";
if (richter >= 3.5)
    r = "Felt by many people, no destruction";
if (richter >= 0)
    r = "Generally not felt by people";

```

In questo modo le condizioni alternative non sono più mutuamente esclusive. Di fatto, se `richter` avesse il valore 6.0, le ultime *quattro* verifiche avrebbero esito positivo e `r` verrebbe impostata quattro volte.



## Argomenti avanzati 5.2

### L'enunciato switch

Per confrontare un singolo valore intero rispetto a diversi valori costanti 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: System.out.print("one"); break;
    case 2: System.out.print("two"); break;
    case 3: System.out.print("three"); break;
    case 4: System.out.print("four"); break;
    case 5: System.out.print("five"); break;
    case 6: System.out.print("six"); break;
    case 7: System.out.print("seven"); break;
    case 8: System.out.print("eight"); break;
    case 9: System.out.print("nine"); break;
    default: System.out.print("error"); break;
}

```

Si tratta di una forma abbreviata per il codice seguente:

```

int digit;
...
if (digit == 1) System.out.print("one");
else if (digit == 2) System.out.print("two");
else if (digit == 3) System.out.print("three");
else if (digit == 4) System.out.print("four");
else if (digit == 5) System.out.print("five");
else if (digit == 6) System.out.print("six");
else if (digit == 7) System.out.print("seven");
else if (digit == 8) System.out.print("eight");
else if (digit == 9) System.out.print("nine");
else System.out.print("error");

```

L'utilizzo dell'enunciato `switch` 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. Possono essere numeri interi, caratteri o enumerazioni (o anche stringhe, in Java 7). Non si può usare `switch` per le diramazioni con numeri in virgola mobile.

Osservate che ciascuna diramazione dell'enunciato `switch` viene terminata da un'istruzione `break`. Se `break` è assente, l'esecuzione procede alla diramazione successiva, ripetutamente, finché raggiunge `break` o la fine dell'enunciato `switch`. Esaminate, ad esempio, questo enunciato:

```
switch (digit)
{
    case 1: System.out.print("one"); // ahi: manca il break
    case 2: System.out.print("two"); break;
    ...
}
```

Se `digit` vale 1, viene eseguito l'enunciato che segue l'etichetta `case 1`:. Dal momento che manca il `break`, viene eseguito anche l'enunciato che segue l'etichetta `case 2`: e il programma stamperà "onetwo".

Esistono alcune situazioni in cui questo comportamento dell'esecuzione, a scorrimento, è effettivamente utile, ma sono molto rare. Peter van der Linden (*Expert C Programming*, Prentice-Hall 1994, pag. 38) riporta un'analisi dell'enunciato `switch` nel codice del compilatore C di Sun. Su 244 enunciati `switch`, ciascuno dei quali presentava una media di sette casi, solo il tre per cento utilizzava il comportamento a scorrimento. Significa che il comportamento normale, vale a dire far procedere l'esecuzione al caso successivo a meno che non ci sia un `break`, è sbagliato nel 97 per cento dei casi. Dimenticarsi di digitare `break` è un errore estremamente diffuso, che genera codice errato.

Lasciamo a voi la scelta di usare o meno l'enunciato `switch` nei vostri programmi. A ogni modo, dovete essere in grado di analizzare e capire un enunciato `switch`, perché potreste incontrarlo nel codice di altri programmatore.

### 5.3.2 Diramazioni annidate

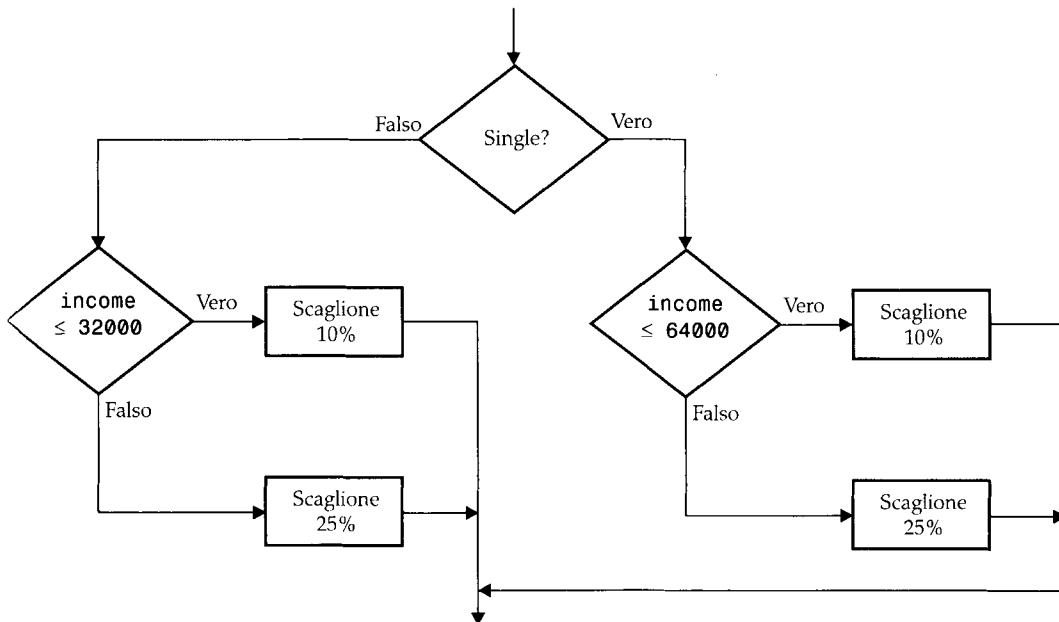
Alcune elaborazioni richiedono decisioni su più *livelli*: prima si prende una decisione, poi almeno una delle strade possibili richiede ulteriori decisioni. Ecco un esempio tipico di questa situazione.

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 congiunta", cumulando i rispettivi redditi e pagando le tasse sul totale. La Tabella 2 riporta i calcoli dell'aliquota d'imposta per ciascuna delle categorie classificate, usando una versione semplificata dei valori relativi alla dichiarazione federale dei redditi del 2008.

Ora calcoliamo le tasse dovute, in base a un determinato stato civile e all'importo del relativo reddito. Per prima cosa dobbiamo scegliere la diramazione relativa allo stato civile; poi, per ciascuno stato civile, dobbiamo imboccare un'altra diramazione in base allo scaglione di reddito (osservate il diagramma di flusso nella Figura 5).

**Tabella 2**  
Aliquote per le tasse federali del 2008  
(versione semplificata)

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 secondo lo schema semplificato) per il 2008

### File ch05/tax/TaxReturn.java

```

/**
 * Una dichiarazione dei redditi di un contribuente per il 2008.
 */
public class TaxReturn
{
    public static final int SINGLE = 1;
    public static final int MARRIED = 2;

    private static final double RATE1 = 0.10;
    private static final double RATE2 = 0.25;
    private static final double RATE1_SINGLE_LIMIT = 32000;
    private static final double RATE1_MARRIED_LIMIT = 64000;

    private double income;
    private int status;
  
```

```

    /**
     * Costruisce una dichiarazione dei redditi per un dato importo
     * del reddito e per un dato stato civile.
     * @param anIncome il reddito del contribuente
     * @param aStatus SINGLE oppure MARRIED
    */
    public TaxReturn(double anIncome, int aStatus)
    {
        income = anIncome;
        status = aStatus;
    }

    public double getTax()
    {
        double tax1 = 0;
        double tax2 = 0;

        if (status == SINGLE)
        {
            if (income <= RATE1_SINGLE_LIMIT)
            {
                tax1 = RATE1 * income;
            }
            else
            {
                tax1 = RATE1 * RATE1_SINGLE_LIMIT;
                tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
            }
        }
        else
        {
            if (income <= RATE1_MARRIED_LIMIT)
            {
                tax1 = RATE1 * income;
            }
            else
            {
                tax1 = RATE1 * RATE1_MARRIED_LIMIT;
                tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
            }
        }

        return tax1 + tax2;
    }
}

```

### File ch05/tax/TaxCalculator.java

```

import java.util.Scanner;

/**
 * Programma che calcola una semplice dichiarazione dei redditi.
 */
public class TaxCalculator
{

```

```

public static void main(String[] args)
{
    Scanner in = new Scanner(System.in);

    System.out.print("Please enter your income: ");
    double income = in.nextDouble();

    System.out.print("Are you married?(Y/N): ");
    String input = in.next();
    int status;
    if (input.equalsIgnoreCase("Y"))
        status = TaxReturn.MARRIED;
    else
        status = TaxReturn.SINGLE;
    TaxReturn aTaxReturn = new TaxReturn(income, status);

    System.out.println("Tax: " + aTaxReturn.getTax());
}
}

```

### Esecuzione del programma

```

Please enter your income: 80000
Are you married?(Y/N): Y
Tax: 10400.0

```

## Auto-valutazione

5. L'enunciato **if/else if/else** per il calcolo degli effetti del terremoto verifica per prima cosa i valori più elevati, poi passa a valori inferiori. Si può invertire quell'ordine?
6. Alcune persone contestano l'applicazione di aliquote più elevate ai redditi più elevati, affermando che, dopo aver pagato le tasse, si potrebbe rimanere con *meno* soldi pur avendo lavorato di più. Qual è l'errore in questo ragionamento?

## 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:

```

if (richter >= 0)
    if (richter <= 4)
        System.out.println("The earthquake is harmless");
    else // Trabocchetto!
        System.out.println("Negative value not allowed");

```

Il livello dei rientri suggerisce che la diramazione **else** sia associata alla verifica **richter >= 0**, ma sfortunatamente non è così. Il compilatore ignora tutti i rientri e segue una regola ben precisa: un **else** appartiene sempre all'**if** più vicino. Ciò significa che il codice, in realtà, è il seguente:

```

if (richter >= 0)
    if (richter <= 4)
        System.out.println("The earthquake is harmless");
    else // Trabocchetto!
        System.out.println("Negative value not allowed");

```

Non è quello che volevamo, perché noi intendevamo associare la diramazione `else` al primo `if`. Per ottenere ciò, dobbiamo usare le parentesi graffe:

```

if (richter >= 0)
{
    if (richter <= 4)
        System.out.println("The earthquake is harmless");
}
else
    System.out.println("Negative value not allowed");

```

Quando il corpo di un `if` contiene un altro `if`, per evitare di doversi preoccupare della giusta corrispondenza delle diramazioni `else`, raccomandiamo di usare *sempre* un paio di parentesi graffe. Nell'esempio seguente le graffe non sono strettamente necessarie, ma rendono il codice più chiaro:

```

if (richter >= 0)
{
    if (richter <= 4)
        System.out.println("The earthquake is harmless");
    else
        System.out.println("Damage may occur");
}

```

L'`else` ambiguo viene detto `else sospeso` (“dangling else”) ed è una tale imperfezione sintattica che alcuni progettisti di linguaggi di programmazione hanno sviluppato una sintassi perfezionata per evitarne completamente il pericolo. Per esempio, il linguaggio Algol 68 usa questa costruzione:

```
if condizione then enunciato else enunciato fi;
```

La parte `else` è facoltativa, ma, poiché la fine dell'enunciato `if` è contrassegnata chiaramente, l'associazione non può essere ambigua, neppure in presenza di due `if` e di un solo `else`. Ecco i due casi possibili:

```
if c1 then if c2 then s1 else s2 fi fi;
```

```
if c1 then if c2 then s1 fi else s2 fi;
```

Fra l'altro, `fi` è proprio `if` scritto a rovescio. Altri linguaggi usano `endif`, che ha la stessa funzione ma è meno buffo.

## Consigli per la produttività 5.2

### 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 fate una colonna per ciascuna variabile. Tenete il codice del programma a portata di mano e usate un contrassegno (come una mollettina) 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 80 000 e la variabile `status` diventa uguale a `MARRIED`. Viene poi invocato il metodo `getTax` e, 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
		6400	4000

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 (\$10 400), dimostra in modo efficace che questo collaudo funziona correttamente.



## Consigli per la produttività 5.3

### Preparare un piano e riservarsi tempo

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 no: in qualità di studenti o di programmatore, 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 potete 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 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 dell'imposta sul reddito, potremmo stimare: 30 minuti per il progetto, perché per la maggior parte è già risolto; 30 minuti per sviluppare casi di prova; un'ora per scrivere il codice al calcolatore e correggere gli errori di sintassi; infine, due ore per il collaudo e per correggere gli errori logici. Il totale è di quattro ore: lavorando a questo progetto due ore al giorno, occorreranno due giorni.

Bisogna prevedere anche le cose che possono andare storte: il computer potrebbe guastarsi, il laboratorio potrebbe essere troppo affollato, 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 conosce il comando “magico” per risolverlo. Quale 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 5.3

### Tipi enumerativi

In molti programmi si usano variabili che possono assumere soltanto un limitato numero di valori. Nell'esempio relativo al 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” oppure “capofamiglia”), potrebbero venire introdotti errori. La versione 5 di Java presenta una soluzione per questi problemi: i *tipi enumerativi*. Una variabile di un tipo

enumerativo può assumere soltanto un valore che appartenga a un insieme appositamente 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 `==`, in questo modo:

```
if (status == FilingStatus.SINGLE) ...
```

Soltanamente le dichiarazioni di tipo enum vengono inserite all'interno di una classe:

```
public class TaxReturn
{
    public TaxReturn(double anIncome, FilingStatus aStatus) { ... }
    ...
    public enum FilingStatus { SINGLE, MARRIED }
    private FilingStatus status;
}
```

Per accedere al tipo enumerativo dall'esterno della classe in cui esso è definito, si usa il nome della classe come prefisso:

```
TaxReturn return = new TaxReturn(income, TaxReturn.FilingStatus.SINGLE);
```

## Sintassi di Java

### 5.3 Dichiarazione di tipo enumerativo

#### Sintassi

```
modalitàDiAccesso enum NomeTipo { valore1, valore2, ... }
```

#### Esempio

Dichiarazione di tipo      `public enum FilingStatus { SINGLE, MARRIED }`

Dichiarazione di variabile      `FilingStatus status;`

Questo variabile può assumere  
(soltanto) i valori  
`FilingStatus.SINGLE,`  
`FilingStatus.MARRIED`  
oppure null.

Una variabile di un tipo enumerativo può anche assumere il valore `null`, cosa che può essere utile per scoprire una variabile non inizializzata o un potenziale errore. Ad esempio, il campo `status` nell'esempio precedente può assumere tre diversi valori: `SINGLE`, `MARRIED` e `null`.

## 5.4 Utilizzare espressioni booleane

### 5.4.1 Il tipo `boolean`

In Java, un'espressione come `amount < 1000` ha un valore, proprio come ha un valore l'espressione `amount + 1000`: il valore di un'espressione relazionale può essere `true` (*vero*) oppure `false` (*falso*). Per esempio, se `amount` è uguale a 500, il valore di `amount < 1000` è `true`. Provate: il seguente frammento di programma visualizzerà `true`.

```
double amount = 0;  
System.out.println(amount < 1000);
```

Il tipo `boolean` ha due possibili valori: `true` e `false`.

I valori `true` e `false` non sono numeri e nemmeno oggetti di una classe: appartengono a un tipo diverso, detto `boolean` (*booleano*), che prende il nome dal matematico George Boole (1815-1864), un pioniere nello studio della logica.

### 5.4.2 I metodi predicativi

Un metodo predicativo restituisce un valore booleano.

Un *metodo predicativo* è un metodo che restituisce un valore di tipo `boolean`; ecco un esempio:

```
public class BankAccount  
{  
    public boolean isOverdrawn()  
    {  
        return balance < 0; // restituisce vero o falso  
    }  
}
```

Il valore restituito dal metodo può essere utilizzato come condizione di un enunciato `if`:

```
if (harrysChecking.isOverdrawn()) ...
```

Nella classe `Character` troviamo parecchi utili metodi predicativi statici:

```
isDigit  
isLetter  
isUpperCase  
isLowerCase
```

che consentono di verificare se un carattere sia, rispettivamente, una cifra, una lettera, una lettera maiuscola o una lettera minuscola:

```
if (Character.isUpperCase(ch)) ...
```

Solitamente il nome di un metodo predicativo inizia, per convenzione, con il prefisso “**is**” oppure “**has**”.

La classe **Scanner** ha metodi predicativi che consentono di verificare se una successiva richiesta di dati in ingresso andrà a buon fine: ad esempio, il metodo **hasNextInt** restituisce **true** se la successiva sequenza di caratteri presente nel flusso identifica un numero intero. Prima di invocare **nextInt**, è opportuno invocare **hasNextInt**:

```
if (in.hasNextInt()) input = in.nextInt();
```

Analogamente, il metodo **hasNextDouble** verifica se una successiva invocazione di **nextDouble** avrà successo.

### 5.4.3 Gli operatori booleani

Immaginate di voler sapere se **amount** è compreso tra 0 e 1000. Devono essere vere due condizioni: **amount** deve essere maggiore di 0 e minore di 1000. In Java si usa l’operatore **&&** per rappresentare la congiunzione *e* (*and*), che combina due verifiche di condizioni. Perciò, la verifica si può scrivere così:

```
if (0 < amount && amount < 1000) ...
```

*Con gli operatori booleani **&&** (*and*), **||** (*or*) e **!** (*not*) si possono comporre verifiche complesse.*

L’operatore **&&** combina più verifiche in una sola, che è vera solamente se tutte le condizioni sono vere. Un operatore che elabora valori booleani è detto *operatore booleano*.

L’operatore **&&** ha una precedenza inferiore a quella degli operatori relazionali, per cui si possono scrivere espressioni relazionali sia a sinistra sia a destra dell’operatore **&&**, senza dover usare parentesi. Ad esempio, nell’espressione

```
0 < amount && amount < 1000
```

vengono prima valutate le espressioni **0 < amount** e **amount < 1000**, dopodiché l’operatore **&&** calcola il risultato. L’Appendice B elenca gli operatori del linguaggio Java e la loro priorità.

Anche l’operatore logico **||** (*or*, che significa *oppure*) combina due o più condizioni: la condizione risultante è vera se almeno una delle condizioni è vera. Ecco, ad esempio, come controllare se la stringa **input** è uguale a “S” o “M”

```
if (input.equals("S") || input.equals("M")) ...
```

La Figura 6 riporta il diagramma di flusso per questi esempi.

Talvolta è necessario *invertire* una condizione, mediante l’operatore logico **!** (*not*, che significa *non*). Ad esempio, possiamo voler eseguire una certa azione solo se due stringhe *non* sono uguali:

```
if (!input.equals("S")) ...
```

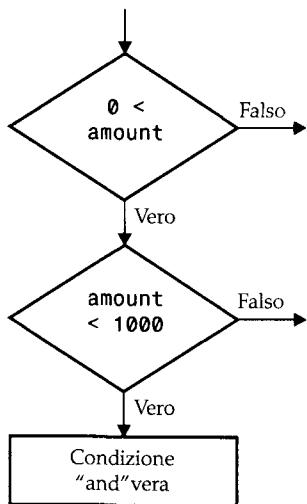
L’operatore **!** esamina una singola condizione e la valuta **true** se è falsa, mentre la valuta **false** se è vera.

Ecco un riepilogo delle tre operazioni logiche:

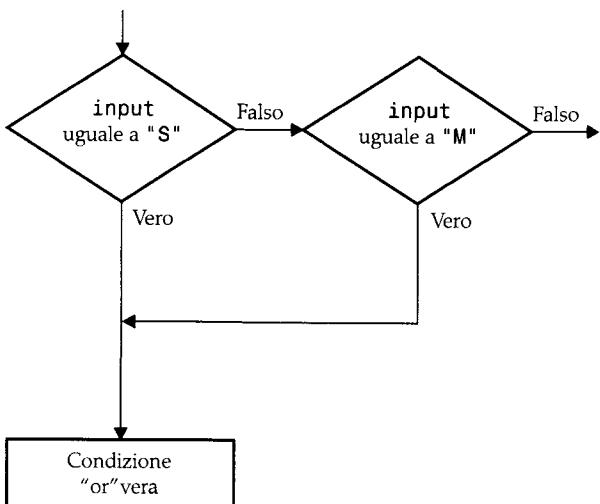
A	B	A && B	A	B	A    B	A	!A
true	true	true	true	qualsiasi valore	true	true	false
true	false	false	false	true	true	false	true
false	qualsiasi valore	false	false	false	false	false	false

**Figura 6**  $0 < \text{amount} \&\& \text{amount} < 1000$

Diagrammi di flusso con operatori  $\&\&$  e  $||$



`input.equals("S") || input.equals("M")`



#### 5.4.4 Utilizzare variabili booleane

Se sapete che una variabile può assumere due soli valori, potete usare una variabile booleana. Date un'altra occhiata al programma per la tassazione visto nel Paragrafo 5.3.2: lo stato civile può essere soltanto “coniugato” oppure no. Invece di usare una variabile intera, si può usare una variabile di tipo boolean:

```
private boolean married;
```

Il risultato della verifica di una condizione può essere memorizzato in una variabile booleana.

Il vantaggio è che risulta impossibile memorizzare per errore un terzo valore nella variabile.

Inoltre, la variabile booleana può essere usata direttamente nella verifica di una condizione:

```
if (married)
  ...
else
  ...
```

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; 100 &lt; 200</code>	Errore di sintassi	<b>Errore.</b> L'espressione <code>0 &lt; 100</code> ha il valore <code>true</code> , che non può essere confrontato con <code>200</code> .
<code>0 &lt; x !! x &lt; 100</code>	true	<b>Errore.</b> Questa condizione è sempre vera. Probabilmente il programmatore voleva scrivere <code>0 &lt; x &amp;&amp; x &lt; 100</code> (si vedano gli Errori comuni 5.5).
<code>0 &lt; x &amp;&amp; x &lt; 100 !! x == -1</code> ( <code>0 &lt; x &amp;&amp; x &lt; 100</code> ) <code>!! x == -1</code>	false	L'operatore <code>&amp;&amp;</code> ha una priorità più elevata dell'operatore <code>!!</code> .
<code>!(0 &lt; 200)</code>	false	<code>0 &lt; 200</code> vale <code>true</code> , quindi il suo inverso è <code>false</code> .
<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>	!frozen	Piuttosto che confrontare con il valore <code>false</code> , l'uso dell'operatore <code>!</code> è più esplicito.

**Tabella 3**  
Operatori booleani

A volte una variabile booleana è detta *flag* (bandiera), perché ha solo due stati, "su" e "giù".

Se pensate attentamente ai nomi da assegnare alle variabili booleane otterrete risultati migliori: nel nostro esempio, non conviene chiamare la variabile booleana `maritalStatus`, perché non ha senso assegnare il valore `true` a uno stato civile. Con un nome come `married` (*sposato*) non vi sono ambiguità: se `married` è `true`, il contribuente è sposato.

Fra l'altro, è considerato goffo scrivere una verifica in questa forma:

```
if (married == true) ... // Da non fare
```

Piuttosto, scrivete semplicemente:

```
if (married) ...
```

Nel Capitolo 6 useremo variabili booleane per controllare cicli complessi.



## Auto-valutazione

7. In quali condizioni l'enunciato seguente visualizza `false`?

```
System.out.println(x > 0 || x < 0);
```

8. Riscrivete l'espressione seguente, evitando di fare il confronto con `false`:

```
if (Character.isDigit(ch) == false) ...
```



## Errori comuni 5.4

### Espressioni con più operatori relazionali

Esaminate l'espressione seguente:

```
if (0 < amount < 1000) ... // Errore
```

Benché l'enunciato assomigli all'espressione matematica “amount è compreso tra 0 e 1000”, purtroppo in Java costituisce un errore di sintassi.

Proviamo ad analizzare la condizione. La prima parte, `0 < amount`, è una verifica con risultato `true` o `false`, in funzione del valore di `amount`. L'esito della verifica (`true` o `false`) viene poi confrontato con 1000: un confronto privo di senso. Il valore `true` è più grande di 1000 oppure no? Si possono confrontare valori di verità e numeri? In Java non si può e il compilatore rifiuta questo enunciato.

Usate, invece, l'operatore `&&` per combinare due verifiche distinte:

```
if (0 < amount && amount < 1000) ...
```

Un altro errore comune, dello stesso genere, è quello di scrivere un enunciato come il seguente, che vorrebbe verificare se `ch` sia uguale a '`S`' o a '`M`':

```
if (ch == 'S' || 'M') ... // Errore
```

Ancora una volta, il compilatore Java segnalerà che questo costrutto è errato, perché non si può applicare l'operatore `||` a valori di tipo carattere. In questo caso, dovete scrivere due espressioni booleane e quindi unirle con l'operatore `||`:

```
if (ch == 'S' || ch == 'M') ...
```

## Errori comuni 5.5

### Confondere le condizioni `&&` e `||`

Confondere le condizioni *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 si possono 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. Le istruzioni per la denuncia dei redditi del 1992 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 alla data del 31 dicembre 1992.
- Eravate vedovi prima del 1 gennaio 1992 e non vi siete risposati nel corso del 1992.

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 vostro coniuge è deceduto nel 1990 o nel 1991, e non vi siete risposati nel 1992.

- Avete un figlio che potete dichiarare a vostro carico.
- Il figlio ha abitato presso di voi per tutto il 1992.
- Avete pagato oltre la metà del costo di gestione della casa per vostro figlio.
- Avete presentato (o avreste potuto presentare) una dichiarazione congiunta insieme con il vostro 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 5.4

### Valutazione "pigra" degli operatori booleani

Gli operatori `&&` e `||` sono calcolati, in Java, mediante valutazione *pigra* (o *cortocircuitata*). In altre parole, le espressioni logiche vengono esaminate da sinistra a destra e l'analisi cessa appena ne 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. Quando viene valutato un operatore *or* e la prima condizione è vera, la seconda condizione non viene esaminata, perché non importa qual è l'esito della seconda verifica. Ecco un esempio:

```
if (input != null && Integer.parseInt(input) > 0) ...
```

Se `input` è `null`, allora la prima condizione è falsa e, quindi, tutto l'enunciato composto è falso, indipendentemente dal risultato della seconda verifica. Se `input` è `null`, la seconda verifica non viene esaminata e si evita il pericolo di scandire una stringa inesistente (un riferimento `null`), azione che solleverebbe un'eccezione.

Se avete, invece, necessità di valutare sempre entrambe le condizioni, allora usate gli operatori `&` e `!` (per i quali potete trovare informazioni nell'Appendice E): quando vengono valutati con operandi booleani, questi operatori esaminano sempre entrambi gli argomenti.



## Argomenti avanzati 5.5

### La legge di De Morgan

Nel paragrafo precedente abbiamo scritto una verifica per controllare se `amount` fosse compreso tra 0 e 1000. Vediamo come verificare se è vero il contrario:

```
if (!(0 < amount && amount < 1000)) ...
```

**La legge di De Morgan indica come semplificare espressioni in cui l'operatore ! sia applicato a termini collegati da operatori && o ||.**

Questa verifica è leggermente più complicata e dovete riflettere attentamente sulla sua logica: "quando *non* è vero che `0 < amount` e che `amount < 1000` ...". Cosa? Come vedete, questo codice può disorientare.

Al computer non importa, ma gli esseri umani generalmente fanno fatica a comprendere le condizioni logiche che applicano operatori *not* a espressioni *and/or*. Per semplificare queste espressioni booleane si può usare la legge di De Morgan, dal nome del logico Augustus De Morgan (1806-1871). La legge di De Morgan si usa in due forme, una per la negazione di un'espressione con *and* e una per la negazione di un'espressione con *or*:

`!(A && B)` è uguale a `!A || !B`  
`!(A || B)` è uguale a `!A && !B`

Fate particolare attenzione al fatto che gli operatori *and* e *or* vengono *scambiati* quando si spostano i *not* all'interno delle parentesi. Ad esempio, la negazione di “`input` è uguale a `S` oppure `input` è uguale a `M`”, cioè

`!(input.equals("S") || input.equals("M"))`

diventa “`input` non è uguale a `S` e `input` non è uguale a `M`”:

`!input.equals("S") && !input.equals("M")`

Proviamo ad applicare questa legge alla negazione di “`amount` è compreso tra 0 e 1000”. Il codice seguente:

`!(0 < amount && amount < 1000)`

equivale a questa verifica:

`!(0 < amount) || !(amount < 1000)`

che può essere ulteriormente semplificata, ottenendo:

`0 >= amount || amount >= 1000`

Notate che l'opposto di `<` è `>=`, non semplicemente `>`!

## 5.5 Collaudo e copertura del codice

Il collaudo a scatola chiusa  
(*black-box testing*) non prende  
in considerazione la struttura  
dell'implementazione.

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.

Tramite white-box testing, volete assicurarvi 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 vostri casi di prova, non avrete modo di sapere se funzionerà correttamente nel caso venga attivata dai dati forniti in ingresso dall'utente. Per raggiungere

Il collaudo trasparente (*white-box testing*) usa informazioni  
sulla struttura del programma.

La copertura di un collaudo misura  
quante parti di un programma siano  
state collaudate.



## Note di cronaca 5.1

### Intelligenza artificiale

Quando si usa un programma informatico sofisticato, come un pacchetto per preparare la dichiarazione dei redditi, si è inclini ad attribuire al computer una forma di intelligenza. Il computer pone quesiti sensati ed esegue calcoli che per noi sarebbero una sfida. Dopo tutto, se fosse facile preparare una denuncia 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 per tutte le eventualità possibili e il software riproduce semplicemente le azioni e le decisioni che sono state programmate.

È possibile scrivere programmi informatici che abbiano una qualche forma di genuina intelligenza? Fin dagli albori dell'informatica, esisteva la percezione che il cervello umano non fosse null'altro che una sorta di enorme computer e che, pertanto, fosse 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 notevoli abilità intellettuali, divennero talmente capaci che oggi sono in grado di battere virtualmente quasi tutti i migliori giocatori umani. Nel 1975, un pro-

gramma del genere *sistema esperto*, chiamato Mycin, si guadagnò la fama di superare la media dei medici nella diagnosi della meningite nei pazienti. I programmi dimostratori di teoremi hanno prodotto prove matematiche logicamente corrette.

Tuttavia, esistono anche serie difficoltà. Dal 1982 al 1992, il governo giapponese si imbarcò in un imponente progetto di ricerca, sovvenzionato con oltre 40 miliardi di yen. Era noto quale *Progetto di quinta generazione* e aveva l'obiettivo di sviluppare nuovo hardware e nuovo software per migliorare enormemente le prestazioni dei sistemi esperti. All'inizio il progetto creò negli altri Paesi il grosso timore che l'industria dei computer giapponese stesse per diventare il *leader* indiscusso nel campo. Tuttavia, i risultati finali furono deludenti e non contribuirono molto a portare sul mercato le applicazioni dell'intelligenza artificiale.

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. Perfino i programmi per il controllo grammaticale, che oggi compaiono in numerosi elaboratori di testi, sono più un macchinegno che uno strumento utile: come se non bastasse, analizzare la

grammatica è solo il primo passo nella traduzione di frasi.

Il progetto CYC (da *encyclopedia*), iniziato da Douglas Lenat nel 1984, è un tentativo di codificare le ipotesi implicite che sottostanno al linguaggio umano, orale e scritto. I membri del gruppo di lavoro partirono dall'analisi degli articoli dei quotidiani e si chiesero quali fossero i fatti non menzionati che risultavano necessari per comprendere effettivamente le frasi. Per esempio, considerate la frase "Last fall she enrolled in Michigan State". Il lettore si rende conto immediatamente che "fall" non significa "declino" o "caduta", ma che, in questo contesto, si riferisce alla stagione dell'anno, "autunno". Sebbene esista uno Stato del Michigan, qui Michigan State indica l'università. A priori, un computer non ha alcuna di queste consapevolezze. L'obiettivo del progetto CYC era quello di estrarre e memorizzare i fatti necessari, ovvero che: 1) le persone si iscrivono alle università; 2) il Michigan è uno Stato; 3) probabilmente, uno Stato X ha un'università, chiamata X State University, spesso abbreviata come X State; 4) la maggior parte delle persone si iscrive all'università in autunno. Nel 1995, il progetto aveva codificato circa 100 000 concetti di senso comune e circa un milione di fatti reali che li mettevano in relazione. Perfino questa imponente quantità di dati si è dimostrata insufficiente per applicazioni utili.

Recentemente, abbiamo assistito a rilevanti passi avanti nelle tecno-

logie guidate dall'intelligenza artificiale. Uno degli esempi più eclatanti riguarda il successo di una serie di "grandi sfide" (*Grand Challenge*) per veicoli autonomi lanciate da DARPA (Defense Advanced Research Projects Agency) negli Stati Uniti. 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: 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. Sebastian Thrun, di Stanford, disse: "Nell'ultimo Grand Challenge, non era importante sapere se un ostacolo fosse una roccia o un cespuglio, perché in ogni caso occorreva aggirarlo. La sfida attuale consiste in un cambiamento di prospettiva: non più semplice analisi dell'ambiente circostante, ma sua comprensione".

questo scopo, dovete esaminare ogni diramazione `if/else`, per controllare che ciascuna venga seguita da qualche caso di prova. Molte diramazioni condizionali 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, perché nessuno ha fornito valori strani o anomali. Naturalmente, questi difetti diventano subito visibili appena si mette all'opera il programma, quando il primo utente inserisce un valore sbagliato e si inferocisce per il fallimento del programma. Un pacchetto di prova dovrebbe garantire 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 in ciascuno dei due scaglioni fiscali.

Quando selezionate i casi di prova, dovreste prendere l'abitudine di considerare anche *casi limite*: valori validi dei dati di ingresso che, però, si trovano al limite dei valori accettabili.

Ad esempio, cosa accade quando vengono calcolate le tasse relative a un reddito pari a zero, oppure quando un conto corrente ha un tasso di interesse uguale allo zero per cento? I casi limite sono valori di ingresso perfettamente leciti, per cui ci si aspetta che il programma li gestisca correttamente, spesso in modo ovvio o mediante alcune condizioni specifiche. Il collaudo dei casi limite è decisamente importante, perché si è constatato che i programmatore tendono a trattarli in modo errato: sintomi tipici di errori nella gestione dei casi limite sono le divisioni per zero, l'estrazione di caratteri da una stringa vuota e l'utilizzo di riferimenti nulli.

I casi limite sono casi di prova che usano valori limite dei dati in ingresso.

## Auto-valutazione

9. Di quanti casi di prova avete bisogno per coprire tutte le diramazioni del metodo `getDescription` della classe `Earthquake`?
10. Indicate un caso limite per collaudare il programma `EarthquakeRunner`. Cosa vi aspettate che venga visualizzato?





## Consigli per la qualità 5.3

### Calcolare manualmente dati di prova

È difficile, e spesso impossibile, dimostrare che un programma funzioni correttamente in qualunque caso. Per assicurarsi della correttezza di un programma, o per capire perché non funziona come dovrebbe, sono preziosissimi i dati di prova calcolati manualmente. Se il programma ottiene gli stessi risultati dei calcoli manuali, la fiducia nel programma viene confermata. In caso contrario, abbiamo un punto di partenza per l'attività di collaudo.

Se solo un programma accenna a svolgere la minima operazione algebrica, molti programmatore sono sorprendentemente restii a eseguire qualsiasi calcolo manuale. A questo atteggiamento contribuisce la loro fobia per la matematica e sperano irrazionalmente di riuscire ad evitare l'algebra e aggirare il problema con tentativi casuali, come un rimescolamento dei segni + e -. I tentativi casuali sono sempre uno sperpero di tempo, che raramente conduce a risultati utili.

Analizziamo nuovamente la classe `TaxReturn`. Nel caso in cui una persona non coniugata abbia un reddito di \$ 50 000, le regole della Tabella 2 indicano che i primi 32 mila dollari vengono tassati con un'aliquota del 10 per cento. Possiamo calcolare mentalmente: la porzione di reddito eccedente i 32 000 dollari viene poi tassata al 25 per cento. È giunto il momento di prendere la calcolatrice: nelle situazioni reali, i numeri sono sempre strani. Calcolate  $(50\ 000 - 32\ 000) \times 0.25 = 4500$ . La tassazione totale è la somma:  $3200 + 4500 = 7700$ . Non è stato poi così difficile.

Eseguite il programma e confrontate i risultati: dato che coincidono, siamo più fiduciosi che il programma sia corretto.

Sarebbe ancora meglio calcolare manualmente alcuni casi di prova prima di scrivere il programma: questa pratica aiuta a comprendere la natura del problema e la sua soluzione, agevolando la realizzazione del programma stesso.

Dovreste sempre calcolare a mano  
alcuni casi di prova, per verificare che  
il vostro programma calcoli  
le risposte corrette.



## Consigli per la qualità 5.4

### Preparare in anticipo i casi di prova

Vediamo come si può collaudare il programma per il calcolo delle imposte. Naturalmente, non possiamo sperimentare tutti i casi di stato civile e tutti i valori di reddito: anche potendo, non avrebbe senso provarli tutti. Se il programma calcola correttamente uno o due carichi fiscali per un certo scaglione di reddito, abbiamo validi motivi per credere che tutti gli importi che ricadono in quello scaglione vengano elaborati correttamente. Noi puntiamo a una *copertura completa* di tutti i casi.

Per lo stato civile ci sono due possibilità e, per ciascuna, due scaglioni di reddito. Questo produce un totale di quattro casi di prova. Poi, vogliamo provare alcune *condizioni limite*, come un reddito pari a zero e un altro che si collochi esattamente sulla soglia tra i due scaglioni. Arriviamo, quindi, a sei casi di prova. Calcolate manualmente le risposte che prevediamo di avere dal programma, come suggerito in Consigli per la qualità 5.3: scriviamo i casi di prova e iniziamo la loro codifica.

Caso di prova	Sposato	Previsto	Commento
30 000	No	3000	Scaglione 10%
72 000	No	13 200	3200 + 25% di 40 000
50 000	Si	5000	Scaglione 10%
104 000	Si	16 400	6400 + 25% di 40 000
32 000	No	3200	Caso limite
0		0	Caso limite

È veramente indispensabile provare sei casi diversi per questo semplice programma? Certamente, è necessario. Inoltre, se nel programma trovate un errore che non viene rilevato dalle prove, create un altro caso e aggiungetelo alla vostra raccolta. Una volta sistemati gli errori identificati, *eseguite nuovamente tutti i casi di prova*: l'esperienza dimostra che, mentre le situazioni che avete appena tentato di sistemare probabilmente funzionano, gli errori corretti in precedenza hanno buone probabilità di ripresentarsi. Se scoprirete che un errore tende a ripresentarsi, di solito è un segno attendibile che non avete afferrato alcune sottili interazioni fra i passi del vostro programma.

Conviene sempre pianificare i casi di prova *prima* di scrivere il codice, per due motivi. Analizzare i casi di prova permette di comprendere meglio l'algoritmo del programma. Inoltre, solitamente i programmatore evitano istintivamente di collaudare le parti meno solide del loro codice. Sembra difficile da credere, ma spesso vi accorgerete di comportarvi allo stesso modo. Osservate qualcun altro mentre verifica il vostro programma: vi saranno situazioni in cui l'altro immetterà dati in ingresso che vi renderanno molto nervosi, perché non siete sicuri che il programma possa gestirli e perché non avete mai osato provarli da soli. Si tratta di un fenomeno ben conosciuto, per il quale stabilire un piano di prove prima di scrivere il codice offre qualche protezione.



## Argomenti avanzati 5.6

### Tracciamento

A volte capita di eseguire un programma e di non capire dove stia perdendo tempo. Per avere un prospetto del flusso di esecuzione potete inserire nel programma messaggi di tracciamento, come questo:

```
public double getTax()
{
    ...
    if (status == SINGLE)
    {
        System.out.println("status is SINGLE");
        ...
    }
    ...
}
```

Tuttavia, visualizzare i messaggi di tracciamento con `System.out.println` pone un problema: quando avete terminato il collaudo del programma, dovete eliminare tutti gli

enunciati di visualizzazione che producono messaggi di tracciamento. Se, però, 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()` 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, mentre riportando 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.

Quando state seguendo il flusso di esecuzione di un programma con il tracciamento, gli eventi più importanti sono l'ingresso in un metodo e l'uscita da esso. All'inizio di un metodo, visualizzatene i parametri:

```
public TaxReturn(double anIncome, int aStatus)
{
    Logger.getGlobal().info("Parameters: anIncome = " + anIncome
                           + " aStatus = " + aStatus);
    ...
}
```

Alla fine di un metodo, visualizzate il valore che verrà restituito:

```
public double getTax()
{
    ...
    Logger.getGlobal().info("Return value = " + tax);
    return tax;
}
```

La classe `Logger` ha molte altre opzioni, per eseguire 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.

## Riepilogo degli obiettivi di apprendimento

### L'enunciato `if` per prendere decisioni

- L'enunciato `if` consente a un programma di compiere azioni diverse in seguito alla verifica di una condizione.
- Un blocco di enunciati raggruppa più enunciati.

### Confronti tra numeri e tra oggetti

- Gli operatori relazionali confrontano valori. L'operatore `==` verifica l'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 l'ordine alfabetico.
- 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

- Si possono combinare più condizioni per prendere decisioni articolate. La loro disposizione corretta dipende dalla logica del problema che si deve risolvere.

### Il tipo di dato `boolean` memorizza il risultato di condizioni: vero o falso

- Il tipo `boolean` ha due possibili valori: `true` e `false`.
- Un metodo predicativo restituisce un valore booleano.
- Con gli operatori booleani `&&` (*and*), `||` (*or*) e `!` (*not*) si possono comporre verifiche complesse.
- Il risultato della verifica di una condizione può essere memorizzato in una variabile booleana.
- La legge di De Morgan indica come semplificare espressioni in cui l'operatore `!` sia applicato a termini collegati da operatori `&&` o `||`.

### Casi di prova per la copertura del codice

- Il collaudo a scatola chiusa (*black-box testing*) non prende in considerazione la struttura dell'implementazione.
- Il collaudo trasparente (*white-box testing*) usa informazioni sulla struttura del programma.
- La copertura di un collaudo misura quante parti di un programma siano state collaudate.
- I casi limite sono casi di prova che usano valori limite dei dati in ingresso.
- Dovreste sempre calcolare a mano alcuni casi di prova, per verificare che il vostro programma calcoli le risposte corrette.

### Il tracciamento mediante la libreria standard può essere attivato e disattivato facilmente

- I messaggi di tracciamento possono essere disattivati dopo aver completato il collaudo.

## Classi, oggetti e metodi presentati nel capitolo

`java.lang.Character`  
`isDigit`  
`isLetter`  
`isLowerCase`  
`isUpperCase`

`java.lang.Object`  
`equals`  
`java.lang.String`  
`equals`  
`equalsIgnoreCase`

```

compareTo          OFF
java.util.Scanner
hasNextDouble      getGlobal
hasNextInt         info
java.util.logging.Level
INFO              setLevel

```

## Esercizi di ripasso

- \* **Esercizio 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;`
- \*\* **Esercizio R5.2.** Trovate gli errori nei seguenti enunciati `if`:
- `if (1 + x > Math.pow(x, Math.sqrt(2)) y = y + x;`
  - `if (x = 1) y++; else if (x = 2) y = y + 2;`
  - `int x = Integer.parseInt(input);  
if (x != null) y = y + x;`
- \*\* **Esercizio R5.3.** Trovate gli errori nel seguente enunciato `if`, che ha l'obiettivo di selezionare una lingua a partire dall'assegnazione di una nazione e di uno stato o provincia.
- ```

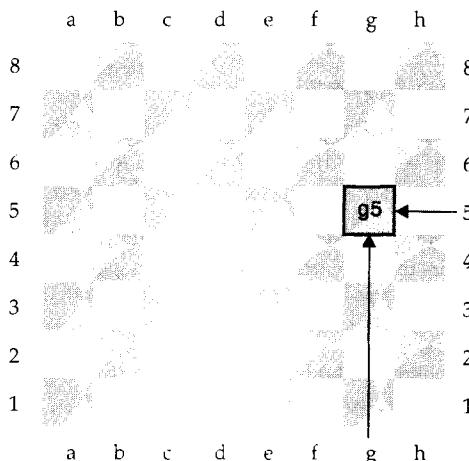
language = "English";
if (country.equals("Canada"))
    if (stateOrProvince.equals(("Quebec")) language = "French";
else if (country.equals("China"))
    language = "Chinese";

```
- \*\* **Esercizio R5.4.** Trovate gli errori nei seguenti enunciati `if`:
- `if (x && y == 0) { x = 1; y = 1; }`
  - `if (1 <= x <= 10)
 System.out.println(x);`
  - `if (!s.equals("nickels") || !s.equals("pennies")
 || !s.equals("dimes") || !s.equals("quarters"))
 System.out.print("Input error!");`
  - `if (input.equalsIgnoreCase("N") || "NO")
 return;`
- \* **Esercizio R5.5.** Spiegate i termini seguenti e fornite un esempio per ciascun costrutto:
- Espressione
  - Condizione
  - Enunciato
  - Enunciato semplice
  - Enunciato composto
  - Blocco (di enunciati)
- \* **Esercizio R5.6.** Spiegate la differenza fra enunciati `if` annidati e un enunciato `if` con più diramazioni `else`. Fornite un esempio per ciascun caso.

- ★ **Esercizio R5.7.** Fornite l'esempio di un enunciato `if/else if/else` in cui l'ordine delle verifiche sia ininfluente. Fornite un altro esempio in cui l'ordine sia invece importante.
- ★ **Esercizio R5.8.** Nelle seguenti coppie di stringhe, qual è quella che precede l'altra nell'ordine lessicografico?
  - "Tom", "Jerry"
  - "Tom", "Tomato"
  - "church", "Churchill"
  - "car manufacturer", "carburetor"
  - "Harry", "hairy"
  - "C++", "Car"
  - "Tom", "Tom"
  - "Car", "Carl"
  - "car", "bar"
  - "101", "11"
  - "1.01", "10.1"
- ★ **Esercizio R5.9.** Completate la seguente tabella dei valori vero e falso (*tabella di verità*), inserendo i valori logici delle espressioni booleane per tutte le combinazioni dei valori booleani p, q e r.

| p     | q     | r                    | $(p \& q) \mid\mid !r$ | $!(p \&& (q \mid\mid !r))$ |
|-------|-------|----------------------|------------------------|----------------------------|
| falso | falso | falso                |                        |                            |
| falso | falso | vero                 |                        |                            |
| falso | vero  | falso                |                        |                            |
|       |       | ...                  |                        |                            |
|       |       | altre 5 combinazioni |                        |                            |
|       |       | ...                  |                        |                            |

- ★★ **Esercizio R5.10.** Ogni casella di una scacchiera può essere descritta da una lettera e un numero, come g5 in questo esempio:



Lo pseudocodice che segue descrive un algoritmo che determina il colore (nero, *black*, o bianco, *white*) di una casella.

Se la lettera è a, c, e oppure g

Se il numero è dispari

colore = "black"

Altrimenti

colore = "white"

Altrimenti

Se il numero è pari

colore = "black"

Altrimenti

colore = "white"

Usando la procedura delineata nei Consigli per la produttività 5.2, eseguite a mano lo pseudocodice con il valore di ingresso **g5**, scrivendo su carta quanto avviene, passo dopo passo.

- \* **Esercizio R5.11.** Progettate quattro casi di prova per l'algoritmo dell'Esercizio precedente, in modo che tutte le sue diramazioni risultino coperte.
- \*\* **Esercizio R5.12.** In un programma di pianificazione, vogliamo verificare se due appuntamenti si sovrappongono. Per semplicità, gli appuntamenti iniziano ad un'ora precisa (cioè con i minuti dell'orario pari a zero) e usiamo la notazione militare, con le ore che vanno da 0 a 23. Lo pseudocodice che segue 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

Usando la procedura delineata nei Consigli per la produttività 5.2, eseguite a mano lo pseudocodice con un appuntamento previsto tra 10 e le 12 e uno tra le 11 e le 13; successivamente, ripetete la procedura con gli appuntamenti 10-11 e 12-13.

- \* **Esercizio R5.13.** Scrivete lo pseudocodice di un programma che chiede all'utente un mese e un giorno e verifica (visualizzando poi un opportuno messaggio) se si tratta di una di queste quattro festività:
  - Capodanno (1 gennaio)
  - Giorno dell'Indipendenza (4 luglio)
  - Giorno dei Veterani (11 novembre)
  - Natale (25 dicembre)
- \*\*\* **Esercizio R5.14.** È vero o falso che **A && B** è sempre uguale a **B && A**, per qualsiasi espressione booleana **A** e **B**?

- \* **Esercizio R5.15.** Spiegate la differenza fra

```
s = 0;
if (x > 0) s++;
if (y > 0) s++;
```

e

```
s = 0;
if (x > 0) s++;
else if (y > 0) s++;
```

- \*\* **Esercizio R5.16.** Utilizzate la legge di De Morgan per semplificare le seguenti espressioni booleans:

- $!(x > 0 \ \&\& \ y > 0)$
- $!(x != 0 \ \|\| \ y != 0)$
- $!(\text{country.equals("US") \ \&\& \ !state.equals("HI")}) \ \&\& \ !\text{state.equals("AK")}$
- $!(x \% 4 != 0 \ \|\| \ !(x \% 100 == 0 \ \&\& \ x \% 400 == 0))$

- \*\* **Esercizio R5.17.** Costruite un nuovo esempio di codice Java per evidenziare il problema dell'`else` sospeso, utilizzando il caso seguente: uno studente con la media scolastica di almeno 1.5 punti, ma inferiore a 2, viene ammesso agli esami, mentre se ha un punteggio inferiore a 1.5 viene bocciato.

- \* **Esercizio R5.18.** Spiegate la differenza fra l'operatore `==` e il metodo `equals` nel confronto di stringhe.

- \*\* **Esercizio R5.19.** Se `r` e `s` sono di tipo `Rectangle`, spiegate la differenza tra le verifiche

```
r == s
```

e

```
r.equals(s)
```

- \*\*\* **Esercizio R5.20.** Quale errore è presente in questo codice, che verifica se `r` è `null`? Cosa succede quando lo si esegue?

```
Rectangle r;
...
if (r.equals(null))
    r = new Rectangle(5, 10, 20, 30);
```

- \* **Esercizio R5.21.** Spiegate perché l'ordinamento lessicografico di stringhe è diverso dall'ordinamento alfabetico che caratterizza un dizionario o un elenco telefonico. Suggerimento: considerate stringhe quali IBM, horstmann.com, Century 21, While-U-Wait, 7-11.

- \*\*\* **Esercizio R5.22.** Scrivete codice Java che verifichi se due oggetti di tipo `Line2D.Double` tracciano la stessa linea quando sono visualizzati sullo schermo grafico. Non usate il costrutto `a.equals(b)`.

```
Line2D.Double a;
Line2D.Double b;

if (inserite qui la vostra condizione)
    g2.drawString("They look the same!", x, y);
```

*Suggerimento:* se  $p$  e  $q$  sono punti, allora `Line2D.Double(p, q)` e `Line2D.Double(q, p)` hanno lo stesso aspetto.

- \* **Esercizio R5.23.** Spiegate perché è più difficile confrontare numeri in virgola mobile rispetto ai numeri interi. Scrivete il codice Java per verificare se un numero intero  $n$  è uguale a 10 e per verificare se un numero  $x$  in virgola mobile è circa uguale a 10.
- \*\* **Esercizio R5.24.** Esaminate il codice seguente, che verifica se un punto cade all'interno di un rettangolo.

```
Point2D.Double p = ...
Rectangle r = ...
boolean xInside = false;
if (r.getX() <= p.getX() && p.getX() <= r.getX() + r.getWidth())
    xInside = true;
boolean yInside = false;
if (r.getY() <= p.getY() && p.getY() <= r.getY() + r.getHeight())
    yInside = true;
if (xInside && yInside)
    g2.drawString("p is inside the rectangle.", p.getX(), p.getY());
```

Riscrivete questo codice per eliminare i valori esplicativi `true` e `false`, assegnando a `xInside` e a `yInside` direttamente valori di espressioni booleane.

- \*T **Esercizio R5.25.** Progettate un insieme di casi di prova per il programma che visualizza la descrizione di terremoti visto nel Paragrafo 5.3.1, garantendo la copertura di tutte le diramazioni.
- \*T **Esercizio R5.26.** Fornite un esempio di caso limite per il programma che visualizza la descrizione di terremoti visto nel Paragrafo 5.3.1. Quale risultato prevedete?

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

# 6

## Iterazioni

### Obiettivi del capitolo

- Saper programmare cicli con gli enunciati `while`, `for` e `do`
- Evitare cicli infiniti ed errori per scarto di uno
- Apprendere gli algoritmi più diffusi che richiedono cicli
- Comprendere il funzionamento dei cicli annidati
- Realizzare simulazioni
- Conoscere il debugger

Questo capitolo presenta i diversi costrutti sintattici del linguaggio Java che consentono di realizzare iterazioni, eseguendo ripetutamente uno o più enunciati finché non viene raggiunto un determinato obiettivo. Vedrete, poi, come applicare le tecniche apprese in questo capitolo per ricevere dati in ingresso e per realizzare simulazioni.

## 6.1 Cicli `while`

In questo capitolo imparerete come scrivere programmi che eseguono ripetutamente uno o più enunciati, presentando questo concetto in relazione a una tipica situazione di investimento economico. Considerate un conto bancario con un saldo iniziale di \$ 10 000 remunerato con un tasso di interesse annuo del 5%. L'interesse viene calcolato e depositato sul conto al termine di ogni anno, basandosi sul saldo di quel momento. Ad esempio, dopo il primo anno si guadagnano \$ 500 (il 5% di \$ 10 000) e tale somma viene accreditata sul conto. L'anno successivo l'ammontare degli interessi è di \$ 525 (il 5% di \$ 10 500) e il saldo diventa \$ 11 025.

Quanti anni occorrono perché il saldo diventi uguale a \$ 20 000? Ovviamente non ci vorranno più di 20 anni, perché ogni anno viene aggiunta al conto bancario la somma di almeno \$ 500, ma potrebbero occorrere meno di 20 anni, perché l'ammontare degli interessi viene calcolato su saldi sempre maggiori. Per calcolare la risposta esatta, scriviamo un programma che aggiunge ripetutamente gli interessi annuali finché non viene raggiunto il saldo previsto.

Un enunciato `while` esegue ripetutamente un blocco di codice, finché una specifica condizione risulta vera.

In Java, l'enunciato `while` realizza questo genere di ripetizione.

```
while (condizione)
    enunciato
```

Questo codice esegue l'*enunciato* finché la *condizione* è vera.

Il più delle volte l'enunciato è un blocco di enunciati, vale a dire una serie di enunciati racchiusa fra parentesi graffe.

Nel nostro caso, vogliamo sapere quando il conto bancario ha raggiunto un certo saldo. Finché il saldo è minore, continuiamo ad aggiungere interessi e a incrementare il contatore `years`:

```
while (balance < targetBalance)
{
    years++;
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

La Figura 1 mostra il flusso d'esecuzione di questo ciclo.

Il codice completo del programma che risolve il nostro problema si trova nella cartella `ch06/invest1` del pacchetto dei file scaricabili per questo libro.

L'enunciato `while` viene spesso chiamato *ciclo*. Se disegnate un diagramma di flusso, vedrete che il controllo, al termine di ciascuna iterazione, torna ciclicamente alla verifica iniziale (osservate la Figura 2).

Quando una variabile viene dichiarata *all'interno* del corpo del ciclo, ad ogni 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)
{
```

```

years++;
double interest = balance * rate / 100;
balance = balance + interest;
} // qui balance non esiste più

```

Se una variabile deve essere aggiornata in più iterazioni del ciclo, bisogna dichiararla all'esterno del ciclo stesso. Ad esempio, non avrebbe senso dichiarare `balance` all'interno di questo ciclo.

**Figura 1**

Esecuzione  
di un ciclo while

- ① Verifica della condizione del ciclo

`balance =`   
`years =`

```

while (balance < targetBalance)
{
    years++;
    double interest = balance * rate / 100;
    balance = balance + interest;
}

```

La condizione  
è vera

- ② Esecuzione degli enunciati interni al ciclo

`balance =`   
`years =`   
`interest =`

```

while (balance < targetBalance)
{
    years++;
    double interest = balance * rate / 100;
    balance = balance + interest;
}

```

- ③ Nuova verifica della condizione del ciclo

`balance =`   
`years =`

```

while (balance < targetBalance)
{
    years++;
    double interest = balance * rate / 100;
    balance = balance + interest;
}

```

La condizione  
è ancora vera

- ④ Dopo 15 iterazioni

`balance =`   
`years =`

```

while (balance < targetBalance)
{
    years++;
    double interest = balance * rate / 100;
    balance = balance + interest;
}

```

La condizione  
non è più vera

- ⑤ Esecuzione dell'enunciato che segue il ciclo

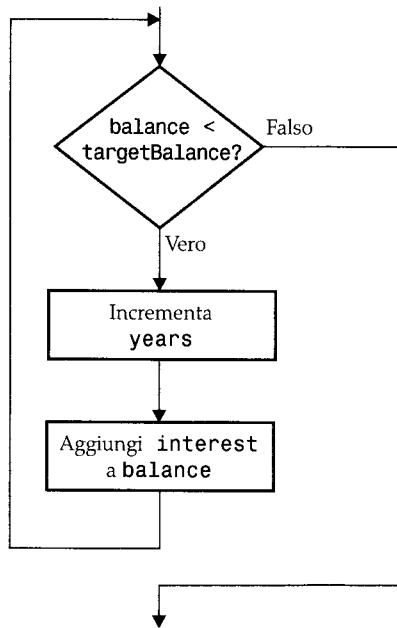
`balance =`   
`years =`

```

while (balance < targetBalance)
{
    years++;
    double interest = balance * rate / 100;
    balance = balance + interest;
}
System.out.println(years);

```

**Figura 2**  
Diagramma di flusso  
di un ciclo while



## Sintassi di Java

### 6.1 L'enunciato while

#### Sintassi

```
while (condizione)
    enunciato
```

#### 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.

È utile incolonnare le parentesi graffe.

```
double balance = 0;
```

...

```
while (balance < TARGET)
```

{

```
    double interest = balance * RATE / 100;
    balance = balance + interest;
```

}

Attenzione agli errori "per scarto di uno" nelle condizioni di terminazione del ciclo.

Non mettere un punto e virgola qui!

Questi enunciati vengono eseguiti finché la condizione è vera.

Se il corpo contiene un solo enunciato, le parentesi graffe non sono necessarie.

Il ciclo seguente esegue ripetutamente l'*enunciato*, senza terminare mai:

```
while (true)
    enunciato
```

A che cosa potrebbe mai servire un programma che non si ferma mai? Esistono due risposte. Effettivamente, alcuni programmi non si fermano mai: il software che controlla uno sportello automatico, un centralino telefonico o un forno a microonde è sempre in esecuzione, almeno finché l'apparecchio non viene spento. Generalmente, i nostri programmi non sono di questo tipo, ma, anche se non siete in grado di terminare il ciclo, potete comunque uscire dal metodo che lo contiene: è una soluzione utile quando la verifica della condizione per la fine delle iterazioni si colloca in modo naturale in un punto interno del ciclo (consultate Argomenti avanzati 6.3).

**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; } i = 0; sum = 0; while (sum &lt; 10) {     i++; sum = sum - i;     Stampa i e sum; } i = 0; sum = 0; while (sum &lt; 0) {     i++; sum = sum - i;     Stampa i e sum; } i = 0; sum = 0; while (sum &gt;= 10) {     i++; sum = sum + i;     Stampa i e sum; } i = 0; sum = 0; while (sum &lt; 10) ; {     i++; sum = sum + i;     Stampa i e sum; }</pre> | <pre>1 1 2 3 3 6 4 10 . . .</pre><br><pre>1 -1 2 -3 3 -6 4 -10 . . .</pre><br><pre>(Niente)</pre><br><pre>(Niente)</pre><br><pre>(Niente,</pre> | <p>Quando <b>sum</b> diventa uguale a 10, la condizione del ciclo diventa falsa e il ciclo termina.</p> <p>Dato che <b>sum</b> non diventa mai uguale a 10, si ha un ciclo infinito (Errori comuni 6.1)</p> <p>La condizione <b>sum &lt; 0</b> è già falsa alla prima verifica, per cui il ciclo non viene mai eseguito.</p> <p>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.</p> <p>Notate il punto e virgola prima della parentesi graffa aperta: questo ciclo ha un corpo vuoto e viene eseguito indefinitamente, continuando a controllare che <b>sum</b> sia minore di 10, senza fare null'altro (Errori comuni 6.4)</p> |



## Auto-valutazione

- Quante volte viene eseguito l'*enunciato* di questo ciclo?  

```
while (false) enunciato;
```
- Cosa succederebbe nel programma `InvestmentRunner` (reperibile nella cartella `ch06/invest1`) se, nel metodo `main`, `RATE` avesse il valore 0?



## Consigli per la produttività 6.1

### Tenere traccia dell'esecuzione di cicli

Nei Consigli per la produttività 5.2 avete visto come eseguire a mano un frammento di codice, tenendone traccia su carta (*hand-tracing*): un metodo particolarmente efficace per capire il funzionamento di un ciclo.

Consideriamo questo semplice ciclo. Cosa visualizza?

```
int n = 1729; // 1
int sum = 0;
while (n > 0) // 2
{
    int digit = n % 10; // 3, 4, 5, 6
    sum = sum + digit;
    n = n / 10;
}
System.out.println(sum); // 7
```

- Vengono usate tre variabili: `n`, `sum` e `digit`. Prima di entrare nel ciclo, le prime due vengono inizializzate, rispettivamente, a 1729 e 0.

| <code>n</code> | <code>sum</code> | <code>digit</code> |
|----------------|------------------|--------------------|
| 1729           | 0                |                    |

- Dato che `n` è positivo, si entra nel ciclo.
- La variabile `digit` assume il valore 9, il resto della divisione per 10 di 1729, mentre alla variabile `sum` viene assegnato il valore  $0 + 9 = 9$ ; infine, `n` diventa uguale a 172 (ricordate che il resto della divisione per 10 di 1729 viene ignorato, perché si tratta di due numeri interi). Tracciate un tratto di penna sui vecchi valori e scrivete, al di sotto, i nuovi.

| <code>n</code> | <code>sum</code> | <code>digit</code> |
|----------------|------------------|--------------------|
| 1729           | 8                |                    |
| 172            | 9                | 9                  |

- Essendo  $n > 0$ , ripetiamo il ciclo. Ora `digit` diventa uguale a 2, a `sum` viene assegnato il valore  $9 + 2 = 11$  e `n` diventa 17.

| n    | sum | digit |
|------|-----|-------|
| 1729 | 8   |       |
| 172  | 9   | 9     |
| 17   | 11  | 2     |

5. Dato che  $n$  non è ancora uguale a zero, ripetiamo nuovamente il ciclo, impostando `digit` a 7, `sum` a  $11 + 7 = 18$  e  $n$  a 1.

| n    | sum | digit |
|------|-----|-------|
| 1729 | 8   |       |
| 172  | 9   | 9     |
| 17   | 11  | 2     |
| 1    | 18  | 7     |

6. Entriamo nel ciclo per l'ultima volta: ora `digit` diventa uguale a 1, `sum` assume il valore 19 e  $n$  diventa zero.

| n    | sum | digit |
|------|-----|-------|
| 1729 | 8   |       |
| 172  | 9   | 9     |
| 17   | 11  | 2     |
| 18   | 18  | 7     |
| 0    | 19  | 1     |

7. La condizione  $n > 0$  è ora falsa, per cui proseguiamo con l'enunciato di visualizzazione che si trova dopo il ciclo: viene visualizzato il numero 19.

È ovvio che la medesima risposta si può ottenere eseguendo semplicemente il codice: la speranza è che, eseguendolo a mano, passo dopo passo, lo possiate *capire*. Vediamo nuovamente cosa accade ad ogni iterazione:

- Estraiamo l'ultima cifra di  $n$ .
- Aggiungiamo tale cifra a `sum`.
- Eliminiamo la cifra da  $n$ .

In altre parole, il ciclo calcola la somma delle cifre che compongono il numero  $n$ : siamo riusciti a capire cosa fa il ciclo per qualsiasi valore di  $n$ , non soltanto nel caso specifico che abbiamo analizzato nell'esempio.

Perché mai qualcuno potrebbe voler sommare le cifre che compongono un numero? Operazioni di questo tipo sono svolte molto frequentemente per verificare la validità dei numeri di carta di credito e di altre forme di numeri identificativi (si veda l'Esercizio P6.2).



## Errori comuni 6.1

### Cicli infiniti

Nei cicli, l'errore più fastidioso è il ciclo infinito: un ciclo che viene eseguito ininterrottamente e che si può fermare solamente annullando l'esecuzione del programma oppure riavviando il computer. Se nel ciclo sono presenti enunciati di visualizzazione, valanghe di righe sfrecciano sullo schermo, altrimenti il programma sembra bloccarsi, come se non stesse facendo niente. In alcuni sistemi potete interrompere un programma che si è bloccato premendo un'apposita combinazione di tasti, come Ctrl+Break o Ctrl+C; in altri potete chiudere la finestra in cui il programma è in esecuzione.

Una causa frequente di cicli infiniti consiste nel dimenticarsi di incrementare la variabile che controlla il ciclo:

```
int years = 0;
while (years < 20)
{
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

In questo codice, il programmatore si è dimenticato di aggiungere al ciclo l'enunciato che incrementa `years`: di conseguenza, il valore di `years` rimane sempre uguale a zero e il ciclo non termina mai.

Un altro errore comune è l'incremento di un contatore che bisognerebbe decrementare, o viceversa. Osservate questo esempio:

```
int years = 20;
while (years > 0)
{
    years++; // Ahi: doveva essere years--
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

In realtà, bisognava decrementare la variabile `years`, non incrementarla. È un errore frequente, perché incrementare i contatori è talmente più naturale che decrementarli, che le vostre dita potrebbero digitare `++` con il pilota automatico. Di conseguenza, `years` risulta essere sempre maggiore di zero e il ciclo non termina mai (in realtà, alla fine il valore di `years` supererà il massimo numero positivo che si può rappresentare, diventando negativo e facendo terminare il ciclo, ma occorre molto tempo e il risultato che si ottiene è completamente errato).



## Errori comuni 6.2

### Errori per scarto di uno

Considerate il nostro calcolo del numero di anni necessari per raddoppiare un investimento:

```
int years = 0;
```

```

while (balance < 2 * initialBalance)
{
    years++;
    double interest = balance * rate / 100;
    balance = balance + interest;
}
System.out.println("The investment reached the target after "
    + years + " years");

```

La variabile `years` deve inizialmente valere zero o uno? La condizione da verificare è `balance < 2 * initialBalance`, oppure `balance <= 2 * initialBalance`? In queste espressioni è facile cadere in un errore *per scarto di uno*.

Qualcuno tenta 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 portare a termine la compilazione e il collaudo di tutte le diverse possibilità: dedicare al problema un piccolo ragionamento fa veramente risparmiare tempo.

| year | balance |
|------|---------|
| 0    | \$ 100  |
| 1    | \$ 150  |
| 2    | \$ 225  |

Fortunatamente, gli errori per scarto di uno si possono evitare facilmente, semplicemente riflettendo con cura su un paio di casi di prova e utilizzando le informazioni ottenute per individuare la logica corretta che va espressa nella condizione del ciclo.

La variabile `years` deve inizialmente valere zero o uno? Ipotizzate uno scenario con valori semplici: un saldo iniziale di \$ 100 e un tasso di interesse del 50%. Dopo un anno, il saldo sarà di \$ 150, e, dopo due anni, di \$ 225, comunque oltre \$ 200. In questo modo, dopo due anni l'investimento è raddoppiato. Quindi, se si esegue due volte il ciclo e se ogni volta si incrementa `years`, significa che `years` deve partire da zero, non da uno.

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: 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 della cifra 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 `years` diventa uguale a uno, mentre `balance` è esattamente uguale a `2 * initialBalance`. Di conseguenza, l'investimento è raddoppiato dopo un anno e *non* bisogna eseguire nuovamente il ciclo. Se la condizione della verifica è `balance < 2 * initialBalance`, il ciclo si ferma, come dovrebbe. Per contro, se la condizione fosse `balance <= 2 * initialBalance`, il ciclo verrebbe eseguito ancora una volta.

In altre parole, dovete continuare a sommare gli interessi finché il saldo *non è ancora raddoppiato*.

Gli errori per scarto di uno sono molto comuni nella programmazione dei cicli: per evitarli, ragionate con cura sui casi più semplici.



## Argomenti avanzati 6.1

### Cicli do

Ci sono casi in cui si vuole eseguire un ciclo almeno una volta e verificare la condizione dopo avere eseguito gli enunciati del corpo. Il ciclo **do** serve a questo:

```
do
    enunciato
  while (condizione);
```

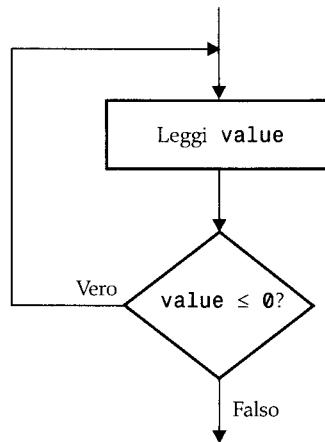
L'*enunciato* viene eseguito finché la *condizione* è vera. Siccome la condizione viene verificata dopo l'esecuzione dell'enunciato, il ciclo viene eseguito almeno una volta.

Per esempio, supponiamo di volere essere sicuri che un utente inserisca un numero positivo. Se l'utente immette un numero negativo o il valore zero, continuiamo semplicemente a richiedere un dato valido. In questa situazione è indicato un ciclo **do**, perché bisogna ricevere un dato dall'utente prima di poterlo esaminare.

```
double value;
do
{
    System.out.print("Please enter a positive number: ");
    value = in.nextDouble();
} while (value <= 0);
```

Osservate il diagramma di flusso.

Diagramma di flusso  
di un ciclo do



In pratica, questa situazione non è molto comune: il codice sorgente della libreria di Java, versione 6, contiene circa 10 000 enunciati di ciclo, ma soltanto il 2 per cento sono cicli **do**. Consideriamo nuovamente l'esempio in cui si vuole chiedere all'utente un numero positivo. In realtà, occorre anche cautelarsi rispetto al caso in cui l'utente introduca un dato che non è un numero, per cui il ciclo diventa più complesso ed è meglio controllarne la terminazione mediante una variabile booleana:

```

boolean valid = false;
while (!valid)
{
    System.out.print("Please enter a positive number: ");
    if (in.hasNextDouble())
    {
        value = in.nextDouble();
        if (value > 0) valid = true;
    }
    else
        in.nextLine(); // elimina i dati non validi
}

```

## 6.2 Cicli for

Il ciclo di gran lunga più comune ha questa forma:

```

i = inizio;
while (i <= fine)
{
    ...
    i++;
}

```

Dal momento che questo ciclo è così comune, esiste una sintassi speciale che ne evidenzia la struttura:

```

for (i = inizio; i <= fine; i++)
{
    ...
}

```

Potete anche dichiarare nell'intestazione del ciclo **for** la variabile usata come contatore nel ciclo: è una comoda scorciatoia, che confina l'utilizzo della variabile all'interno del corpo del ciclo (come sarà discusso ulteriormente in Argomenti avanzati 6.2).

```

for (int i = inizio; i <= fine; i++)
{
    ...
}

```

Possiamo usare questo tipo di ciclo per determinare il saldo finale del nostro investimento di \$ 10 000 con un tasso di interesse del 5% per una durata di 20 anni, saldo che sarà ovviamente maggiore di \$ 20 000, perché ogni anno vengono aggiunti almeno \$ 500. Potreste essere sorpresi nel vedere di quanto sia maggiore il saldo.

Nel nostro ciclo, la variabile **i** si incrementa da 1 a **numberOfYears**, il numero di anni per i quali vogliamo calcolare l'interesse composto.

```

for (int i = 1; i <= numberOfYears; i++)
{
}

```

**Si usa un ciclo for quando una variabile varia da un valore iniziale a un valore finale con un incremento o decremento costante.**

```

        double interest = balance * rate / 100;
        balance = balance + interest;
    }
}

```

La Figura 3 mostra il diagramma di flusso corrispondente, mentre nella Figura 4 trovate l'analisi dettagliata dell'esecuzione, passo dopo passo.

Il ciclo `for` viene spesso usato anche per esaminare tutti i caratteri di una stringa:

```

for (int i = 0; i < str.length(); i++)
{
    char ch = str.charAt(i);
    Elabora ch
}

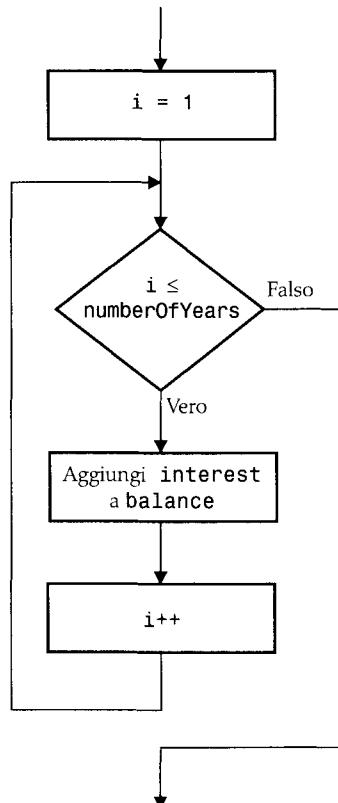
```

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, `i` assume i valori 0, 1, 2, 3 e 4: sono proprio le posizioni valide dei caratteri presenti nella stringa.

Notate anche che le tre sezioni dell'intestazione del ciclo `for` possono contenere espressioni qualsiasi. Ad esempio, potete contare all'indietro, invece che in avanti:

```
for (int i = 10; i > 0; i--)
```

**Figura 3**  
Diagramma di flusso  
di un ciclo for



**Figura 4**

Esecuzione di un ciclo for

## (1) Inizializzazione del contatore

```
i = 
for (int i = 1; i <= numberOfYears; i++)
{
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

## (2) Verifica della condizione

```
i = 
for (int i = 1; i <= numberOfYears; i++)
{
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

## (3) Esecuzione del corpo del ciclo

```
i = 
for (int i = 1; i <= numberOfYears; i++)
{
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

## (4) Aggiornamento del contatore

```
i = 
for (int i = 1; i <= numberOfYears; i++)
{
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

## (5) Nuova verifica della condizione

```
i = 
for (int i = 1; i <= numberOfYears; i++)
{
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

Poi, non è necessario che l'incremento (o il decremento) procedano di un'unità alla volta:

```
for (int i = -10; i <= 0; i = i + 2) ...
```

ed è anche possibile, benché sia segno di incredibile cattivo gusto, inserire nell'intestazione del ciclo condizioni estranee:

```
for (rate = 5; years-- > 0; System.out.println(balance))
... // Di pessimo gusto
```

Non vogliamo nemmeno tentare di decifrare che cosa significhi questo enunciato: dovreste usare soltanto cicli **for** che inizializzano, verificano e aggiornano un'unica variabile.

**Sintassi di Java****6.2 L'enunciato for****Sintassi**

```
for (inizializzazione; condizione; aggiornamento)
    enunciato
```

**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.

| Ciclo                              | Valori di i                                                 | Commento                                                                                                   |
|------------------------------------|-------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|
| for (i = 0; i <= 5; i++)           | 0 1 2 3 4 5                                                 | Il ciclo viene eseguito 6 volte (Consigli per la qualità 6.4).                                             |
| 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).          |

**Tabella 2 File ch06/invest2/Investment.java**

Esempi con cicli for

```
/*
 * Una classe che controlla la crescita di un investimento che
 * accumula interessi a un tasso annuale fisso.
 */
public class Investment
{
```

```
private double balance;
private double rate;
private int years;

/***
 * Costruisce un oggetto di tipo Investment
 * con un saldo iniziale e un tasso di interesse.
 * @param aBalance il saldo iniziale
 * @param aRate il tasso d'interesse percentuale
 */
public Investment(double aBalance, double aRate)
{
    balance = aBalance;
    rate = aRate;
    years = 0;
}

/***
 * Continua ad accumulare interessi finché il saldo
 * non raggiunge un valore desiderato.
 * @param targetBalance il saldo desiderato
 */
public void waitForBalance(double targetBalance)
{
    while (balance < targetBalance)
    {
        years++;
        double interest = balance * rate / 100;
        balance = balance + interest;
    }
}

/***
 * Continua ad accumulare interessi per
 * il numero di anni richiesto.
 * @param n il numero di anni
 */
public void waitYears(int numberOfYears)
{
    for (int i = 1; i <= numberOfYears; i++)
    {
        double interest = balance * rate / 100;
        balance = balance + interest;
    }
    years = years + numberOfYears;
}

/***
 * Restituisce il saldo attuale dell'investimento.
 * @return il saldo attuale
 */
public double getBalance()
{
    return balance;
}
```

```

    /**
     Restituisce il numero di anni per i quali l'investimento
     ha accumulato interessi.
     @return il numero di anni trascorsi dall'inizio dell'investimento
 */
public int getYears()
{
    return years;
}
}

```

### File ch06/invest2/InvestmentRunner.java

```

    /**
     Questo programma calcola di quanto aumenta
     un investimento in un numero di anni assegnato.
 */
public class InvestmentRunner
{
    public static void main(String[] args)
    {
        final double INITIAL_BALANCE = 10000;
        final double RATE = 5;
        final int YEARS = 20;
        Investment invest = new Investment(INITIAL_BALANCE, RATE);
        invest.waitYears(YEARS);
        double balance = invest.getBalance();
        System.out.printf("The balance after %d years is %.2f\n",
                           YEARS, balance);
    }
}

```

### Esecuzione del programma

The balance after 20 years is 26532.98



### Auto-valutazione

3. Riscrivete, sotto forma di ciclo `while`, il ciclo `for` presente nel metodo `waitYears`.
4. Quante volte viene eseguito questo ciclo `for`?

```

for (i = 0; i <= 10; i++)
    System.out.println(i * i);

```



### Consigli per la qualità 6.1

#### Usare i cicli `for` solamente per lo scopo previsto

Un ciclo `for` è un'espressione *idiomatica* che rappresenta un ciclo `while` di forma particolare. Si attiva un contatore che viene modificato da un valore iniziale a un valore finale con incrementi costanti:

```
for (imposta contatore al valore inizio;
```

```

    verifica se contatore ha il valore fine;
    aggiorna contatore secondo l'incremento)
{
    ...
    // contatore, inizio, fine, incremento
    // non dovrebbero essere modificati qui
}

```

Se il vostro ciclo non ha questa struttura, non usate il costrutto `for`. Ricordate, però, che il compilatore non può impedirvi di scrivere cicli `for` stupidi:

```

// pessimo stile: espressioni estranee nell'intestazione
for (System.out.println("Inputs: "));
    (x = in.nextDouble()) > 0;
    sum = sum + x)
    count++;

for (int i = 1; i <= years; i++)
{
    if (balance >= targetBalance)
        i = years; // pessimo stile: modifica il contatore
    else
    {
        double interest = balance * rate / 100;
        balance = balance + interest;
    }
}

```

Questi cicli funzionano, ma sono indubbiamente scadenti. Per le iterazioni che non si adattano alla struttura del ciclo `for`, utilizzate un ciclo `while`.

## Errori comuni 6.3

### Dimenticare un punto e virgola

A volte succede che tutte le operazioni di un ciclo vengano già svolte nella sua intestazione. Se ignoraste i Consigli per la qualità 6.1, per raddoppiare l'investimento potreste scrivere un ciclo come questo:

```

for (years = 1;
    (balance = balance + balance * rate / 100) < targetBalance;
    years++)
;
System.out.println(years);

```

Il corpo del ciclo `for` è completamente vuoto: contiene solo un enunciato vuoto, terminato da un punto e virgola.

Se scrivete un ciclo senza corpo, è importante che siate ben sicuri di non dimenticare il punto e virgola: se lo omettete accidentalmente, la linea successiva diventa parte integrante dell'enunciato del ciclo!

```

for (years = 1;
    (balance = balance + balance * rate / 100) < targetBalance;

```

```
    years++)
System.out.println(years);
```

Per far risaltare bene il punto e virgola, inseritelo in una riga a sé stante, come mostrato nel primo esempio.



## Errori comuni 6.4

### Un punto e virgola di troppo

Cosa visualizza il ciclo seguente?

```
sum = 0;
for (i = 1; i <= 10; i++);
    sum = sum + i;
System.out.println(sum);
```

Naturalmente, si immagina che il ciclo calcoli  $1 + 2 + \dots + 10 = 55$ . In realtà, l'enunciato finale visualizza 11!

Perché 11? Diamo un altro sguardo. Riuscite a vedere il punto e virgola alla fine del ciclo `for`? In realtà, questo ciclo ha un corpo vuoto.

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

Il ciclo non fa nulla per dieci volte e, quando termina, `sum` vale ancora 0, mentre `i` è uguale a 11. Poi, il programma esegue l'enunciato:

```
sum = sum + i;
```

assegnando 11 a `sum`. L'enunciato aveva un rientro verso destra e ingannava il lettore, però il compilatore ignora i rientri.

Il punto e virgola alla fine dell'enunciato è, ovviamente, un errore di battitura. A volte le dita sono talmente abituate a digitare un punto e virgola alla fine di ciascuna riga, da aggiungerlo per sbaglio anche al ciclo `for`. Il risultato è un ciclo con il corpo vuoto.



## Consigli per la qualità 6.2

### Non usare l'operatore != per verificare la fine di un intervallo

Ecco un ciclo che nasconde un'insidia:

```
for (i = 1; i != n; i++)
```

La condizione `i != n` è pessima: che cosa succederebbe se `n` fosse negativo o uguale a zero? La condizione `i != n` non sarebbe mai falsa, perché `i` inizia da 1 e si incrementa.

La soluzione è semplice. Usate `<=` invece di `!=`:

```
for (i = 1; i <= n; i++) ...
```



## Argomenti avanzati 6.2

### Visibilità delle 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 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, avete due o più condizioni di uscita, potreste avere ancora bisogno della variabile. Per esempio, considerate questo ciclo:

```
for (i = 1; balance < targetBalance && i <= n; i++)
{
    ...
}
```

In pratica, volete che il saldo raggiunga un valore prestabilito, ma siete disposti ad aspettare al massimo un numero di anni prefissato. Se il saldo raggiunge il valore desiderato in anticipo, potreste 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 inserire 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 Consigli per la qualità 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--;
}
```



## Consigli per la qualità 6.3

### Limiti simmetrici e asimmetrici

È facile scrivere un ciclo in cui *i* varia da 1 a *n*:

```
for (i = 1; i <= n; i++) ...
```

I valori di *i* sono delimitati dalla relazione  $1 \leq i \leq n$ . Dal momento che su entrambi i lati ci sono operatori di confronto del tipo  $\leq$ , i limiti sono detti *simmetrici*.

Quando si esaminano tutti i caratteri presenti in una stringa, i limiti sono *asimmetrici*.

```
for (i = 0; i < s.length(); i++) ...
```

**Nei cicli, valutate l'esigenza di limiti simmetrici o asimmetrici.**

I valori di *i* sono delimitati dalla relazione  $0 \leq i < s.length()$ , con un operatore di confronto  $\leq$  sulla sinistra e con il segno  $<$  sulla destra: è giusto, perché *s.length()* non è una posizione valida.

Non conviene forzare una simmetria in modo artificioso:

```
for (i = 0; i <= s.length() - 1; i++) ...
```

perché l'enunciato diventa più difficile da leggere e da capire.

In ciascun ciclo, valutate qual è la forma più naturale secondo le esigenze del problema, e usate quella.



## Consigli per la qualità 6.4

### Contare le iterazioni

Individuare il limite inferiore e superiore di un ciclo può non essere banale. Bisogna iniziare da zero? Come condizione di terminazione, è meglio usare  $\leq b$  oppure  $< b$ ?

Per comprendere meglio un ciclo è molto utile contare il numero di iterazioni, operazione che risulta essere più agevole per i cicli con limiti asimmetrici. Questo ciclo viene eseguito *b* - *a* volte

```
for (i = a; i < b; i++) ...
```

Per esempio, questo ciclo, che esamina i caratteri di una stringa, viene eseguito *str.length()* volte

```
for (i = 0; i < str.length(); i++) ...
```

Ciò è perfettamente logico, perché nella stringa esistono *str.length()* caratteri.

Questo ciclo, che ha limiti simmetrici

```
for (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 (n = 0; n <= 10; n++)
```

Può darsi che questo sia l'effetto desiderato; se non lo è, iniziate da 1 oppure usate la condizione  $< 10$ .

**Contate il numero di iterazioni di un ciclo, per verificare che sia corretto.**

Un metodo per scoprire l'errore dovuto a questo “+1” è quello di immaginare i pali e le sezioni di una staccionata. Se la staccionata ha dieci sezioni (=), quanti pali (!) deve avere?

```
!=|=|=|=|=|=|=|=|=|=|=|
```

Una staccionata con dieci sezioni ha *undici* pali, perché ciascuna sezione ha un palo sulla sinistra e c'è un ulteriore palo a destra dell'ultima sezione. Quando ci si dimentica di contare l'ultima iterazione di un ciclo espresso con la condizione  $\leq$  si parla di “errore del palo di staccionata”.

Se il valore dell'incremento,  $c$ , è diverso da 1, i conteggi delle iterazioni diventano:

|                       |                       |
|-----------------------|-----------------------|
| ( $b - a$ ) / $c$     | per cicli asimmetrici |
| ( $b - a$ ) / $c + 1$ | per cicli simmetrici  |

Per esempio, il ciclo `for (i = 10; i <= 40; i += 5)` viene eseguito  $(40 - 10) / 5 + 1 = 7$  volte.

## 6.3 Algoritmi comuni che usano 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.

### 6.3.1 Calcolo di un totale

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. Ovviamenete, deve essere inizializzata a zero.

```
double total = 0;
while (in.hasNextDouble())
{
    double input = in.nextDouble();
    total = total + input;
}
```

### 6.3.2 Conteggio di eventi

Spesso si vuol sapere quanti valori soddisfano una particolare condizione: ad esempio, quante lettere maiuscole sono presenti in una stringa. Si usa un *contatore*: una variabile che viene inizializzata a zero e incrementata ogni volta che la condizione è soddisfatta.

```
int upperCaseLetters = 0;
for (int i = 0; i < str.length(); i++)
{
    char ch = str.charAt(i);
    if (Character.isUpperCase(ch))
    {
        upperCaseLetters++;
    }
}
```

Se, ad esempio, `str` è la stringa "Hello, World!", `upperCaseLetters` viene incrementata due volte (quando `i` vale 0 e 7).

### 6.3.3 Identificazione della prima corrispondenza

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 la prima lettera minuscola all'interno di una stringa. Date che non esaminiamo tutti i caratteri della stringa, invece di un ciclo `for` è meglio usare un ciclo `while`:

```
boolean found = false;
char ch = '?';
int position = 0;
while (!found && position < str.length())
{
    ch = str.charAt(position);
    if (Character.isLowerCase(ch)) { found = true; }
    else { position++; }
}
```

Se si trova un valore che soddisfa la condizione, `found` assume il valore `true`, `ch` contiene il primo carattere che corrisponde a quanto si cercava e la sua posizione nella stringa è 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()`.

Si noti che la variabile `ch` è stata dichiarata *al di fuori* del ciclo `while`, perché la vogliamo usare dopo che il ciclo è terminato.

### 6.3.4 Richiesta ripetuta fino al raggiungimento di un obiettivo

Nell'esempio precedente abbiamo cercato, all'interno di una stringa, un carattere che soddisfacesse una determinata condizione: la medesima procedura può essere applicata alla lettura dei dati forniti in ingresso dall'utente. Immaginate di voler chiedere all'utente

di inserire un valore positivo minore di 100. Continuate a chiedere finché non viene fornito un valore corretto:

```
boolean valid = false;
double input = 0;
while (!valid)
{
    System.out.println("Please enter a positive value < 100: ");
    input = in.nextDouble();
    if (0 < input && input < 100) { valid = true; }
    else { System.out.println("Invalid input. "); }
}
```

Come nell'esempio precedente, la variabile `input` è stata dichiarata al di fuori del ciclo `while`, per poterla usare dopo che il ciclo è terminato.

### 6.3.5 Confronto di valori adiacenti

Quando in un ciclo si elabora una sequenza di valori, a volte si ha la necessità di confrontare un valore con 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:

```
double input = 0;
while (in.hasNextDouble())
{
    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.

Risposta: dobbiamo 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"); }
}
```

Ci 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 un'operazione di lettura iniziale, 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"); }
    }
}

```

### 6.3.6 Lettura di dati mediante valori “sentinella”

Immaginate di voler elaborare un insieme di valori, per esempio un insieme di misurazioni, per calcolarne alcune proprietà, come la loro media o il loro valore massimo. Per prima cosa chiedete all’utente di digitare il primo valore, poi il secondo, poi il terzo e così via. Quando saranno terminati i dati in ingresso?

Un metodo molto utilizzato per segnalare la fine di un insieme di dati prevede l’uso di un *valore sentinella*, un valore che non fa parte dell’insieme dei dati validi: la presenza della sentinella indica che i dati sono terminati.

Alcuni programmati scelgono, come valori sentinella, numeri come 0 oppure -1, ma non sempre questa è una buona idea, perché tali valori possono essere validi come dati del problema. Un’idea migliore consiste nell’utilizzo di un dato d’ingresso che non sia un numero, come la lettera Q (“Quit”, fine), come in questo tipico esempio di esecuzione di un programma:

```

Enter value, Q to quit: 1
Enter value, Q to quit: 2
Enter value, Q to quit: 3
Enter value, Q to quit: 4
Enter value, Q to quit: Q
Average = 2.5
Maximum = 4.0

```

Ovviamente, in questo caso dobbiamo leggere ciascun dato in ingresso sotto forma di stringa, non di numero. Dopo aver verificato che quanto è stato letto non sia la lettera Q, convertiamolo in numero.

```

System.out.print("Enter value, Q to quit: ");
String input = in.next();
if (input.equalsIgnoreCase("Q"))
    abbiamo finito
else
{
    double x = Double.parseDouble(input);
    ...
}

```

Adesso abbiamo un altro problema: la verifica della condizione di terminazione del ciclo avviene *in un punto interno* al ciclo, non all’inizio o alla fine. Per prima cosa dovete tentare di leggere un numero, poi potrete verificare se i dati in ingresso sono terminati. In Java, non esiste una struttura di controllo preconfezionata che segua lo schema “esegui una parte del lavoro, fai una verifica, quindi esegui un’altra parte del lavoro”: dobbiamo usare una struttura formata da un ciclo while e da una variabile di tipo boolean.

A volte la condizione di terminazione  
di un ciclo può essere valutata  
soltanto all’interno del ciclo stesso,  
che può essere controllato mediante  
una variabile booleana.

```

boolean done = false;
while (!done)
{
    visualizza una richiesta di dati
}

```

```
String input = leggi un dato;
if (i dati sono terminati)
    done = true;
else
{
    elabora il dato letto
}
}
```

Questa struttura viene talvolta chiamata “ciclo e mezzo”. Alcuni programmatore trovano scomodo inserire una variabile di controllo in un ciclo di questo tipo: la sezione Argomenti avanzati 6.3 presenta alcune alternative.

Scriviamo ora un programma che legge e analizza un insieme di valori. Per separare la lettura dei dati in ingresso dal calcolo delle caratteristiche dei dati che ci interessano, useremo due classi: `DataAnalyzer` e `DataSet`. La classe `DataAnalyzer` legge i dati e li inserisce in un oggetto di tipo `DataSet`, invocandone il metodo `add`; successivamente, ne invoca i metodi `getAverage` e `getMaximum`, per ottenere, rispettivamente, il valore medio e il valore maggiore di tutti i dati inseriti.

### File ch06/dataset/DataAnalyzer.java

```
import java.util.Scanner;

/**
 * Questo programma calcola il valore medio e il valore massimo
 * in un insieme di dati forniti in ingresso.
 */
public class DataAnalyzer
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        DataSet data = new DataSet();

        boolean done = false;
        while (!done)
        {
            System.out.print("Enter value, Q to quit: ");
            String input = in.next();
            if (input.equalsIgnoreCase("Q"))
                done = true;
            else
            {
                double x = Double.parseDouble(input);
                data.add(x);
            }
        }

        System.out.println("Average = " + data.getAverage());
        System.out.println("Maximum = " + data.getMaximum());
    }
}
```

**File ch06/dataset/DataSet.java**

```

    /**
     * Calcola informazioni relative a un insieme di dati.
    */
public class DataSet
{
    private double sum;
    private double maximum;
    private int count;

    /**
     * Costruisce un insieme di dati vuoto.
    */
    public DataSet()
    {
        sum = 0;
        count = 0;
        maximum = 0;
    }

    /**
     * Aggiunge un valore all'insieme di dati.
     * @param x un valore
    */
    public void add(double x)
    {
        sum = sum + x;
        if (count == 0 || maximum < x) maximum = x;
        count++;
    }

    /**
     * Restituisce la media dei valori inseriti.
     * @return la media, oppure 0 se non sono stati inseriti dati
    */
    public double getAverage()
    {
        if (count == 0) return 0;
        else return sum / count;
    }

    /**
     * Restituisce il valore massimo tra i valori inseriti.
     * @return il massimo, oppure 0 se non sono stati inseriti dati
    */
    public double getMaximum()
    {
        return maximum;
    }
}

```

**Esecuzione del programma**

```

Enter value, Q to quit: 10
Enter value, Q to quit: 0

```

```
Enter value, Q to quit: -1
Enter value, Q to quit: Q
Average = 3.0
Maximum = 10.0
```

## Auto-valutazione

5. Come si calcola la somma di tutti i numeri positivi ricevuti in ingresso?
6. Come si comporta l'algoritmo del Paragrafo 6.3.5 quando non riceve alcun dato in ingresso? Come si risolve questo problema?
7. Perché la classe `DataAnalyzer` invoca `in.next` e non `in.nextDouble`?
8. Semplificando nel modo seguente l'aggiornamento del campo `maximum` nel metodo `add`, la classe `DataSet` calcolerebbe il valore massimo in modo ugualmente corretto?

```
if (maximum < x) maximum = x;
```

## Consigli pratici 6.1

### Realizzare cicli

Questa sezione vi guida attraverso il procedimento che occorre seguire quando si programma un ciclo. Illustreremo passo dopo passo la soluzione di questo problema: leggi dodici valori di temperatura (uno per ogni mese) e visualizza il numero che indica il mese che ha fatto registrare la temperatura più elevata. Ad esempio, nel sito <http://worldclimate.com>, si legge che il valore medio mensile delle temperature massime nella Valle della Morte è (mese per mese):

```
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 deve visualizzare 7.

**Fase 1** Decidete cosa deve essere fatto *all'interno* del ciclo

Ogni ciclo deve svolgere qualche compito ripetitivo, come

- Leggere un altro dato in ingresso.
- Aggiornare un valore (come un saldo bancario o un totale).
- Aggiornare un contatore.

Se non riuscite a individuare cosa debba essere fatto all'interno del ciclo, iniziate a scrivere, una dopo l'altra, le cose che fareste per risolvere il problema a mano. Ad esempio, in merito al problema della lettura delle temperature, potremmo scrivere

Leggi il primo valore.

Leggi il secondo valore.

Se il secondo valore è maggiore del primo, imposta la temperatura massima a tale valore; il mese con la temperatura massima assume il valore 2.

Leggi il valore successivo.

Se tale valore è maggiore del primo e del secondo, imposta la temperatura massima a tale nuovo valore; il mese con la temperatura massima assume il valore 3.

Leggi il valore successivo.

Se tale valore è maggiore della temperatura massima registrata finora, imposta la temperatura massima a tale nuovo valore; il mese con la temperatura massima assume il valore 4.

...

A questo punto, esaminate ciò che avete scritto e identificate un insieme di azioni *ripetitive* che possano essere inserite nel corpo del ciclo. La prima di tali azioni è semplice:

Leggi il valore successivo.

L'azione successiva è più complessa. Nella nostra descrizione, abbiamo usato le condizioni "maggiore del primo", "maggiore del primo e del secondo" e "maggiore della temperatura massima registrata finora": dobbiamo, invece, individuare una condizione che funzioni per tutte le iterazioni del ciclo, ma l'ultima di queste è, chiaramente, la più generale ed è, quindi, quella che ci serve.

Analogamente, dobbiamo trovare un modo generale per impostare il valore del mese avente la temperatura massima: ci serve una variabile che memorizzi il valore del mese attuale, assumendo valori che vanno da 1 a 12. Possiamo quindi così formulare la seconda azione del ciclo:

Se tale valore è maggiore della temperatura massima registrata finora, imposta la temperatura massima a tale nuovo valore; il mese con la temperatura massima assume il valore corrispondente al mese attuale.

Mettendo tutto insieme, il nostro ciclo diventa:

Ripeti

Leggi il valore successivo.

Se tale valore è maggiore della temperatura massima registrata finora, imposta la temperatura massima a tale nuovo valore; il mese con la temperatura massima assume il valore corrispondente al mese attuale.

Incrementa il mese attuale.

### Fase 2 Determinate la condizione che controlla il ciclo

Quale obiettivo volete raggiungere con il vostro ciclo? Ecco alcuni esempi tipici:

- Il contatore ha raggiunto il suo valore finale?
- Abbiamo letto l'ultimo valore presente in ingresso?
- C'è un valore che ha raggiunto la soglia prevista?

Nel nostro esempio, vogliamo semplicemente che il mese attuale arrivi al valore 12.

### Fase 3 Decidete il tipo di ciclo

Distinguiamo i cicli in due categorie principali. Un ciclo *definito* o *controllato da un contatore* viene eseguito un numero predefinito di volte, mentre per un ciclo *indefinito* o *controllato*

*da un evento* il numero di iterazioni non è noto a priori: il ciclo termina quando si verifica un certo evento. Un esempio tipico di questa seconda categoria è un ciclo che legge dati fino all'arrivo di una sentinella.

I cicli definiti sono ben realizzati da un ciclo **for**. Quando, invece, vi trovate di fronte a un ciclo indefinito, prendete in esame la condizione di terminazione: riguarda valori che vengono impostati soltanto all'interno del corpo del ciclo? In tal caso dovreste scegliere un ciclo **do**, per essere certi che il ciclo venga eseguito almeno una volta prima che la condizione di terminazione venga valutata, altrimenti usate un ciclo **while**.

Altre volte la condizione di terminazione di un ciclo cambia valore in un punto intermedio del corpo del ciclo stesso, per cui è bene usare una variabile booleana che indichi quando è giunto il momento di terminare il ciclo, seguendo questo schema:

```
boolean done = false;
while (!done)
{
    fai alcune cose
    if (tutto ciò che doveva essere fatto è stato fatto)
    {
        done = true;
    }
    else
    {
        fa altre cose
    }
}
```

Una variabile di questo tipo viene chiamata *flag*.

Riassumendo:

- Se sapete a priori il numero di iterazioni del ciclo, usate un ciclo **for**.
- Se il 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 Impostate le variabili necessarie per eseguire la prima iterazione

Elencate tutte le variabili che vengono utilizzate o aggiornate all'interno del ciclo e individuate come inizializzarle. Solitamente i contatori vengono inizializzati a 0 oppure a 1, mentre i totali partono da zero.

Nel nostro esempio le variabili sono:

me<sup>se</sup> attuale  
temperatura massima  
me<sup>se</sup> con temperatura massima

Dobbiamo fare attenzione al valore iniziale che assegniamo alla variabile “temperatura massima”, perché non possiamo banalmente impostarlo a zero: in fin dei conti, il nostro programma deve poter funzionare anche con valori di temperatura registrati nell’Antartide, che saranno tutti negativi.

In questo caso, usare come primo valore di “temperatura massima” il primo valore di temperatura letto in ingresso è una buona idea. Ovviamente, dobbiamo poi ricordarci di leggere soltanto altri 11 valori, facendo partire il “mese attuale” dal valore 2.

Occorre anche inizializzare a 1 il “mese con temperatura massima”: dopotutto, in una città australiana sarebbe arduo trovare un mese più caldo di gennaio!

#### Fase 5 Dopo la terminazione del ciclo, elaborate i valori prodotti

In molti casi, il risultato desiderato è semplicemente contenuto in una delle variabili che sono state oggetto di aggiornamenti all'interno del ciclo: è proprio questo il caso del nostro esempio, dove il risultato è il “mese con temperatura massima”. Altre volte, invece, il ciclo calcola valori che contribuiranno alla determinazione del risultato finale. Immaginate, infatti, di voler calcolare la media delle temperature registrate: in questo caso, il ciclo calcolerà la somma delle temperature, non la loro media, che verrà calcolata soltanto al termine del ciclo, dividendo la somma per il numero di dati letti.

Ecco, infine, il nostro ciclo completo.

```

Leggi il primo valore e memorizzalo come temperatura massima.
mese con temperatura massima = 1
for (mese attuale = 2; mese attuale <= 12; mese attuale++)
    Leggi il valore successivo.
    Se tale valore è maggiore della temperatura massima
        temperatura massima = valore appena letto
        mese con temperatura massima = mese attuale
    
```

#### Fase 6 Eseguite il ciclo a mano con alcuni valori d'esempio

Eseguite il ciclo a mano, su carta, come descritto nei Consigli per la produttività 6.1. Scegliete, come esempi, casi che non siano troppo complicati: l'esecuzione di quattro o cinque iterazioni è sufficiente per verificare la presenza della maggior parte degli errori più frequenti. Controllate con la massima cura la prima e l'ultima iterazione eseguita.

A volte sarà utile semplificare un po' il problema per rendere ragionevole l'esecuzione manuale. Ad esempio, volendo tracciare su carta un'esecuzione del problema del raddoppio dell'investimento, si potrebbe usare un tasso di interesse del 20 per cento, anziché il 5% originario. Nel nostro ciclo, relativo alle temperature, usiamo soltanto 4 dati, invece di 12.

Ipotizziamo che i dati siano 22.6, 36.6, 44.5 e 24.2. Ecco la traccia dell'esecuzione:

| Mese attuale | Valore letto | Mese con temperatura massima | Temperatura massima |
|--------------|--------------|------------------------------|---------------------|
|              |              | X                            | 22.6                |
| 2            | 36.6         | X                            | 36.6                |
| 3            | 44.5         | 3                            | 44.5                |
| 4            | 24.2         |                              |                     |

Come si vede, le variabili “mese con temperatura massima” e “temperatura massima” sono state inizializzate ai valori corretti.

**Fase 7** Realizzate il ciclo in Java

Ecco il ciclo completo per il nostro esempio. L'Esercizio P6.1 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;
    }
}
```



## Esempi completi 6.1

### Elaborazione di carte di credito

Quando si fanno acquisti in Internet, molti siti chiedono di inserire il numero della carta di credito senza spazi e senza 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.

Vogliamo rimuovere tutti gli spazi e i trattini presenti nella stringa `creditCardNumber`: ad esempio, se la stringa fosse "4123-5678-9012-3450", dovremmo generare "4123567890123450".

**Fase 1** Decidete cosa deve essere fatto *all'interno* del ciclo

Nel ciclo, ispezioniamo tutti i caratteri della stringa, uno alla volta, 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 = il carattere in posizione `i` in `creditCardNumber`

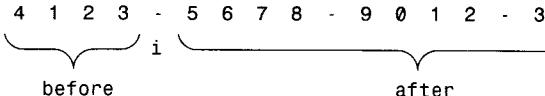
    Se `ch` è uno spazio o un trattino

        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`, occorre concatenare le due sottostringhe che terminano prima di `i` e iniziano 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*: 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 Determinate la condizione che controlla il ciclo

Continuiamo ad eseguire il ciclo finché l'indice *i* fa riferimento a una posizione valida, cioè

```
i < creditCardNumber.length()
```

#### Fase 3 Decidete il tipo di ciclo

Sappiamo a priori quante iterazioni del ciclo verranno eseguite, perché ogni carattere della stringa originaria verrà esaminato una e una sola volta, ma non si tratta di un vero e proprio ciclo a contatore, perché la variabile di controllo del ciclo, *i*, non viene incrementata ad ogni iterazione: sceglieremo, quindi, 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 Impostate le variabili necessarie per eseguire la prima iterazione

È molto semplice: dobbiamo porre *i* = 0.

#### Fase 5 Dopo la terminazione del ciclo, elaborate i valori prodotti

In questo caso, il risultato prodotto è proprio la stringa che ci serve, per cui possiamo scrivere lo pseudocodice del nostro ciclo completo.

```
i = 0
Finché i < creditCardNumber.length()
    ch = il carattere in posizione i in creditCardNumber
    Se ch è uno spazio o un trattino
        Elimina il carattere da creditCardNumber
    Altrimenti
        Incrementa i
```

#### Fase 6 Eseguite il ciclo a mano con alcuni valori d'esempio

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.

```
public class CCNumber
{
    public static void main(String[] args)
    {
        String creditCardNumber = "4123-5678-9012-3450";

        int i = 0;
        while (i < creditCardNumber.length())
        {
            char ch = creditCardNumber.charAt(i);
            if (ch == ' ' || ch == '-')
            {
                String before = creditCardNumber.substring(0, i);
                String after = creditCardNumber.substring(i + 1);
                creditCardNumber = before + after;
            }
            else
            {
                i++;
            }
        }

        System.out.println(creditCardNumber);
    }
}
```

## Argomenti avanzati 6.3

### Il problema del "ciclo e mezzo"

Per leggere dati in ingresso, siamo abituati a usare un ciclo simile a questo, che è piuttosto brutto:

```
boolean done = false;
while (!done)
```

```

{
    String input = in.next();
    if (input.equalsIgnoreCase("Q"))
        done = true;
    else
    {
        elabora il dato
    }
}

```

La verifica effettiva per determinare l'uscita dall'iterazione si trova in un punto interno al ciclo, non all'inizio. Questa struttura è chiamata “ciclo e mezzo”, perché bisogna spingersi fino a metà di un'iterazione del ciclo per sapere se è giunto il momento di terminarlo.

Alcuni programmatore detestano profondamente aggiungere un'ulteriore variabile booleana per controllare il ciclo. Per ridurre il problema del “ciclo e mezzo”, si possono usare due caratteristiche del linguaggio Java: non penso che rappresentino una soluzione migliore, ma, dal momento che entrambe le alternative sono piuttosto comuni, vale la pena di conoscerle per leggere il codice di altre persone.

Nella condizione di terminazione del ciclo, potete combinare un'assegnazione e una verifica:

```

while (!(input = in.next()).equalsIgnoreCase("Q"))
{
    elabora il dato
}

```

L'espressione

```
(input = in.next()).equalsIgnoreCase("Q")
```

significa “per prima cosa leggi un dato (mediante `in.next()`), poi assegna il dato alla variabile `input` e, infine, verifica se è stata letta la sentinella”. È un'espressione con un effetto collaterale. Lo scopo principale dell'espressione è di fungere da condizione per il ciclo `while`, ma, in realtà, svolge anche una parte di lavoro, ovvero legge il dato e lo inserisce nella variabile `input`. Generalmente, è sempre una pessima idea avvalersi di effetti collaterali, perché rendono il programma difficile da leggere e da aggiornare. In questo caso, tuttavia, è una soluzione piuttosto allettante, perché elimina la variabile di controllo `done`, che, a sua volta, rende difficoltosa la lettura e la manutenzione del codice.

L'altra soluzione consiste nell'uscire dal ciclo a metà, mediante un enunciato `return` o un enunciato `break` (Argomenti avanzati 6.4).

```

public void processInput(Scanner in)
{
    while (true)
    {
        String input = in.next();
        if (input.equalsIgnoreCase("Q"))
            return;
        elabora il dato
    }
}

```



## 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`. L'enunciato `break` dell'esempio seguente termina il ciclo quando si raggiunge la fine dei dati in ingresso.

```
while (true)
{
    String input = in.next();
    if (input.equalsIgnoreCase("Q"))
        break;
    double x = Double.parseDouble(input);
    data.add(x);
}
```

In generale, usare un enunciato `break` per uscire da un ciclo è una soluzione molto scadente. Nel 1990, il cattivo uso di un `break` causò un guasto in un commutatore telefonico 4ESS di AT&T, che si propagò attraverso tutta la rete degli Stati Uniti, rendendola pressoché inutilizzabile per circa nove ore. Un programmatore aveva inserito un `break` per terminare un enunciato `if`: sfortunatamente, non ha senso usare `break` in un `if`, quindi l'esecuzione del programma abbandonò il blocco dell'enunciato `switch` che conteneva l'enunciato `if`, impedendo l'inizializzazione di alcune variabili e gettando il sistema nel caos [Expert C Programming, Peter van den Linden, Prentice-Hall 1994, pag. 38]. L'uso di enunciati `break` rende difficile anche l'impiego delle tecniche per la *dimostrazione di correttezza*, che vedrete in Argomenti avanzati 6.5.

Nonostante questi rischi, a fronte della scomodità di dover inserire una variabile dedicata al solo 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`, che si utilizza per uscire da un enunciato annidato. L'enunciato `break etichetta`; fa proseguire l'esecuzione del programma immediatamente dalla fine dell'enunciato contrassegnato 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`, munito di etichetta, fu inventato 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;
```

```

    }
}

prosegue da questo punto se è successo qualcosa di veramente brutto

```

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 dell'*iterazione attuale* di un ciclo. Ecco un possibile utilizzo di questo enunciato:

```

while (!done)
{
    String input = in.next();
    if (input.equalsIgnoreCase("Q"))
    {
        done = true;
        continue; // salta alla fine del corpo del ciclo
    }
    double x = Double.parseDouble(input);
    data.add(x);
    // l'enunciato continue provoca la prosecuzione da questo punto
}

```

Se si usa l'enunciato `continue`, non occorre inserire all'interno di una clausola `else` la parte rimanente del codice del ciclo. Tuttavia, questo è un vantaggio trascurabile e pochi programmatore utilizzano questo enunciato.

## 6.4 Cicli annidati

I cicli possono essere annidati. Un esempio tipico di ciclo annidato è quello che visualizza tabelle con righe e colonne.

A volte il corpo di un ciclo è un altro ciclo: in questo caso diciamo che il ciclo più interno è *annidato* nel ciclo più esterno e ciò accade spesso, ad esempio, quando si vogliono elaborare strutture bidimensionali, come le tabelle.

Vediamo un esempio che ci pare un po' più interessante di una tabella di numeri. Supponiamo di dover stampare la seguente forma triangolare:

```

[]
[[]]
[[[]]]
[[[],[]]]
[[[],[],[]]]
[[[],[],[],[]]]
[[[],[],[],[],[]]]

```

L'idea di base è semplice. Dobbiamo stampare una serie di righe:

```

for (int i = 1; i <= width; i++)
{
    // una riga del triangolo
    ...
}

```

Come stampare una riga del triangolo? Usiamo un altro ciclo per mettere insieme le parentesi quadre che compongono la riga, quindi aggiungiamo un carattere per andare a

capo alla fine della riga. La riga *i*-esima ha *i* simboli, per cui il contatore del ciclo varia tra 1 e *i*.

```
for (int j = 1; j <= i; j++)
    r = r + "[ ]";
r = r + "\n";
```

Se si mettono insieme entrambi i cicli, si ottengono due *cicli annidati*:

```
String r = "";
for (int i = 1; i <= width; i++)
{
    // una riga del triangolo
    for (int j = 1; j <= i; j++)
        r = r + "[ ]";
    r = r + "\n";
}
return r;
```

**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>                                                                           | ****<br>****<br>****     | 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>                                                                           | ***<br>***<br>***<br>*** | 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>       | -*-<br>-*-<br>-*-        | 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 + j) % 2 == 0) { Visualizza "*" }         else { Visualizza " " }     }     System.out.println(); }</pre> | * * *<br>* *<br>* * *    | Visualizza uno schema a scacchiera.                                        |



## Auto-valutazione

9. Come modifichereste i cicli annidati in modo da visualizzare un quadrato, invece di un triangolo?

10. Qual è il valore di `n` dopo l'esecuzione dei seguenti cicli annidati?

```
int n = 0;
for (int i = 1; i <= 5; i++)
    for (int j = 0; j < i; j++)
        n = n + j;
```



## 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. Per realizzare questa classe si 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 (troverete la classe nella cartella `ch06/image` del pacchetto di file scaricabili per questo libro):

```
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) { ... }

    /**
     * ...
     */
}
```

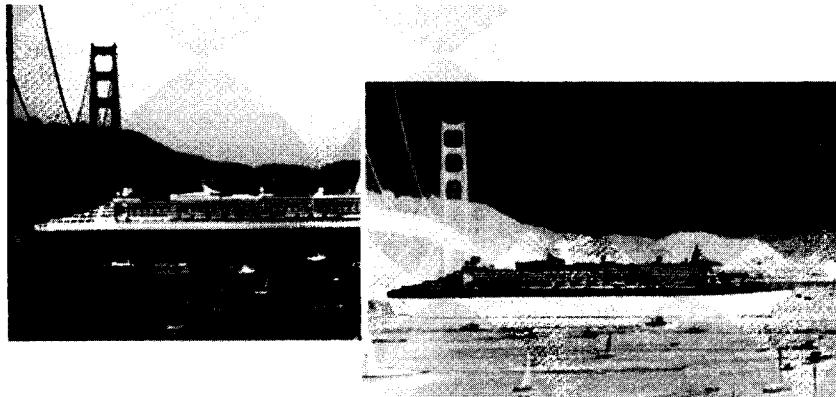
```

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) { ... }

...
}

```

Abbiamo ora il compito di 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 telecamere.



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, svolgere la medesima elaborazione con tutti i pixel che compongono 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. Essendoci `pic.getWidth()` colonne, 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. Essendoci `pic.getHeight()` righe, 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:

```
import java.awt.Color;

public class Negative
{
    public static void main(String[] args)
    {
        Picture pic = new Picture();
        pic.load("queen-mary.png");
        for (int x = 0; x < pic.getWidth(); x++)
            for (int y = 0; y < pic.getHeight(); y++)
            {
                Color original = pic.getColorAt(x, y);
                Color negative = new Color(255 - original.getRed(),
   255 - original.getGreen(),
   255 - original.getBlue());
                pic.setColorAt(x, y, negative);
            }
    }
}
```

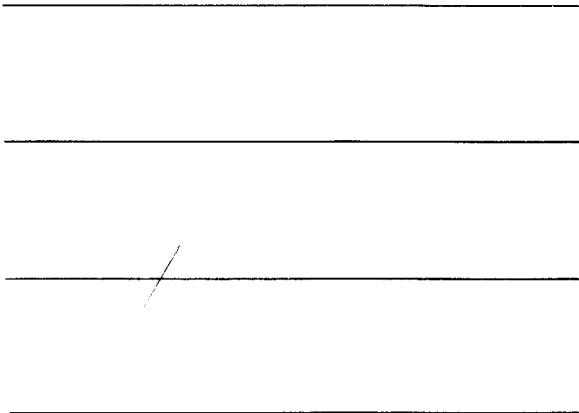
## 6.5 Numeri casuali e simulazioni

In una simulazione si generano ripetutamente numeri casuali, usandoli per simulare una determinata attività.

In un *programma di simulazione*, usate il computer per simulare un'attività del mondo reale (o di un mondo immaginario). Le simulazioni sono molto utilizzate per prevedere i mutamenti climatici, per analizzare il traffico stradale, per decidere investimenti finanziari e per molte altre applicazioni scientifiche e industriali. In molte simulazioni si usano cicli per modificare ripetutamente lo stato di un sistema e osservare i conseguenti cambiamenti.

**Figura 5**

L'esperimento dell'ago  
di Buffon



Ecco un tipico problema che si può risolvere eseguendo una simulazione: l'*esperimento dell'ago di Buffon*, ideato dal conte Georges-Louis Leclerc de Buffon (1707-1788), naturalista francese. A ogni *tentativo* si lascia cadere un ago, lungo un pollice, su un foglio di carta sul quale sono tracciate righe spaziate di due pollici l'una dall'altra. Se l'ago cade su una riga, si conta un bersaglio colpito (osservate la Figura 5). Buffon ipotizzò che il quoziente fra *tentativi* e *successi* fosse prossimo a  $\pi$ .

In che modo potete riprodurre questo esperimento al calcolatore? Non volete assolutamente costruire un robot che lancia aghi sulla carta! La classe `Random` della libreria di Java realizza un *generatore di numeri casuali*, che produce numeri apparentemente del tutto arbitrari. Per generare numeri casuali, costruite un oggetto della classe `Random` e poi applicate uno dei metodi seguenti:

| 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) |

Per esempio, potete simulare il lancio di un dado in questo modo:

```
Random generator = new Random();
int d = 1 + generator.nextInt(6);
```

L'invocazione `generator.nextInt(6)` restituisce un numero intero casuale compreso fra zero e cinque (inclusi): per ottenere un numero fra uno e sei, aggiungiamo uno.

Se invocate `nextInt` dieci volte, ottenete una sequenza di numeri casuali simile a questa:

```
6 5 6 3 2 6 3 4 4 1
```

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, spesso vengono chiamati numeri *pseudocasuali*. 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`.

Per riprodurre l'esperimento dell'ago di Buffon dobbiamo faticare ancora un po'. Quando un dado viene lanciato, deve presentare una delle sue sei facce, mentre quando lanciate un ago possono accadere molte situazioni diverse. Dovete generare *due* numeri casuali per ogni lancio: uno per descrivere la posizione della punta dell'ago e uno per descrivere l'angolo dell'ago rispetto all'asse  $x$ . Poi, dovete verificare se l'ago tocca una linea della griglia. Fermatevi dopo 10 000 tentativi.

Decidiamo di generare la posizione dell'estremità *inferiore* dell'ago. La sua coordinata  $x$  è irrilevante, mentre possiamo stabilire che la sua coordinata  $y$ , indicata da  $y_{\text{low}}$ , corrisponda a un numero casuale compreso fra zero e due. Dal momento che può trattarsi di un *numero in virgola mobile*, useremo il metodo `nextDouble` della classe `Random`, che restituisce un numero casuale in virgola mobile compreso fra zero e uno. Moltiplicheremo il risultato per due, per ottenere un numero casuale compreso fra zero e due.

L'angolo  $\alpha$ , misurato fra l'ago e l'asse  $x$ , può avere qualsiasi valore compreso fra 0 e 180 gradi. La coordinata  $y_{\text{high}}$  dell'estremità superiore dell'ago si calcola nel modo seguente:

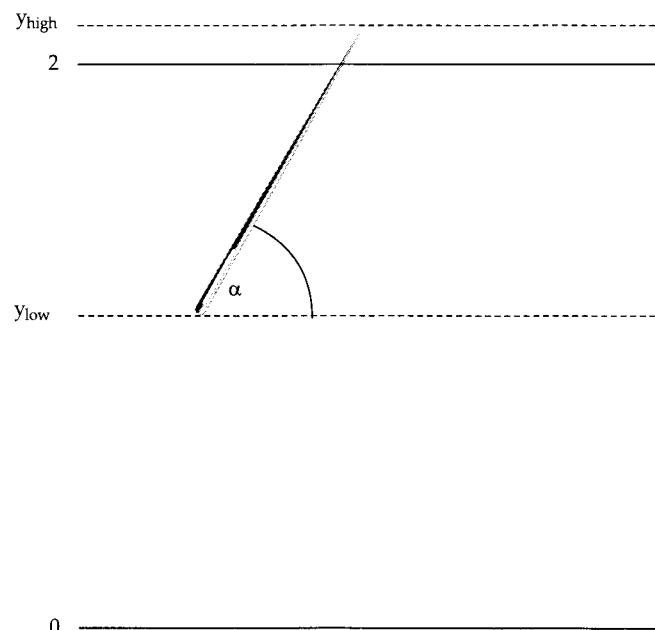
$$y_{\text{high}} = y_{\text{low}} + \text{sen}(\alpha)$$

Come si vede nella Figura 6, si colpisce il bersaglio quando il valore di  $y_{\text{high}}$  è almeno uguale a 2.

Potete trovare il programma per eseguire la simulazione dell'esperimento nella cartella `ch06/random2` del pacchetto dei file scaricabili per questo libro.

Lo scopo di questo programma *non* è il calcolo di  $\pi$ , dal momento che esistono modi molto più efficienti per farlo, bensì di dimostrare come si possa riprodurre al computer un esperimento fisico. Buffon doveva lasciar effettivamente cadere l'ago per migliaia di volte

**Figura 6**  
L'ago cade sulla griglia?



e annotare il risultato, un'attività piuttosto tediosa. Voi potete far eseguire l'esperimento al computer, in modo rapido e accurato.

Le simulazioni sono applicazioni del computer molto comuni. Tutte le simulazioni usano essenzialmente lo stesso schema visto per questo esempio: all'interno di un ciclo, si genera una grande quantità di valori da usare in prova; per ciascuna prova, si conservano i valori di alcune osservazioni; completata la simulazione, si stampano le medie (o altri parametri statistici) dei valori rilevati.

Un esempio tipico di simulazione è il modello delle code di clienti in banca o al supermercato. Invece di osservare i clienti veri, si simulano al computer i loro arrivi e le loro operazioni allo sportello o al banco della cassa di uscita. Al computer si possono sperimentare disposizioni diverse del personale o delle strutture fisiche dell'edificio, semplicemente modificando il programma: apportare tutte queste modifiche nel mondo reale e valutare il loro impatto sarebbe impossibile, o, perlomeno, molto dispendioso.

## Auto-valutazione

11. Come si può usare un generatore di numeri casuali per simulare il lancio di una moneta?
12. Perché l'esecuzione del programma `NeedleSimulator` non è un modo efficiente per il calcolo di  $\pi$ ?

## Argomenti avanzati 6.5

### Condizione invariante in un ciclo

Se  $a$  è un numero in virgola mobile e  $n$  è un numero intero positivo, considerate il problema di calcolare  $a^n$ . Naturalmente, potete moltiplicare  $a \times a \times \dots \times a$ ,  $n$  volte, ma, se  $n$  è un numero elevato, vi troverete a fare molte moltiplicazioni. Il ciclo seguente calcola  $a^n$  mediante un numero di passi molto minore:

```
double a = ...;
int n = ...;
double r = 1;
double b = a;
int i = n;
while (i > 0)
{
    if (i % 2 == 0) // i è pari
    {
        b = b * b;
        i = i / 2;
    }
    else
    {
        r = r * b;
        i--;
    }
}
// qui r è uguale alla n-esima potenza di a
```

Considerate il caso  $n = 100$ . Il metodo esegue questi calcoli:

| b        | i   | r         |
|----------|-----|-----------|
| a        | 100 | 1         |
| $a^2$    | 50  |           |
| $a^4$    | 25  |           |
|          | 24  | $a^4$     |
| $a^8$    | 12  |           |
| $a^{16}$ | 6   |           |
| $a^{32}$ | 3   |           |
|          | 2   | $a^{36}$  |
| $a^{64}$ | 1   |           |
|          | 0   | $a^{100}$ |

In maniera abbastanza sorprendente, l'algoritmo restituisce esattamente  $a^{100}$ . Avete capito perché? Vi siete convinti che funzionerà per qualsiasi valore di  $n$ ? Esiste un metodo ingegnoso per dimostrare che questo algoritmo restituirà sempre il risultato corretto. Dimostreremo che, ogni volta che il programma raggiunge l'inizio del ciclo `while`, la seguente condizione è vera:

$$r \cdot b^i = a^n \quad (I)$$

Sicuramente la condizione è soddisfatta nella prima iterazione, perché  $r = 1$ ,  $b = a$  e  $i = n$ . Supponiamo, quindi, che la condizione (I) sia soddisfatta all'inizio del ciclo. Contrassegneremo i valori di  $r$ , di  $b$  e di  $i$  con il pedice “old” quando entriamo nel ciclo e con il pedice “new” quando ne usciamo. Quindi, per ipotesi:

$$r_{\text{old}} \cdot b_{\text{old}}^{i_{\text{old}}} = a^n$$

Nel ciclo dobbiamo distinguere i due casi in cui  $i_{\text{old}}$  è pari o dispari. Se  $i_{\text{old}}$  è pari, il ciclo effettua le trasformazioni seguenti:

$$\begin{aligned} r_{\text{new}} &= r_{\text{old}} \\ b_{\text{new}} &= b_{\text{old}}^2 \\ i_{\text{new}} &= i_{\text{old}}/2 \end{aligned}$$

Di conseguenza:

$$\begin{aligned} r_{\text{new}} \cdot b_{\text{new}}^{i_{\text{new}}} &= r_{\text{old}} \cdot (b_{\text{old}})^{2^{i_{\text{old}}/2}} \\ &= r_{\text{old}} \cdot b_{\text{old}}^{i_{\text{old}}} \\ &= a^n \end{aligned}$$

D'altra parte, se  $i_{\text{old}}$  è dispari, abbiamo:

$$r_{\text{new}} = r_{\text{old}} \cdot b_{\text{old}}$$

$$b_{\text{new}} = b_{\text{old}}$$

$$i_{\text{new}} = i_{\text{old}} - 1$$

Per cui:

$$\begin{aligned} r_{\text{new}} \cdot b_{\text{new}}^{i_{\text{new}}} &= r_{\text{old}} \cdot b_{\text{old}} \cdot b_{\text{old}}^{i_{\text{old}} - 1} \\ &= r_{\text{old}} \cdot b_{\text{old}}^{i_{\text{old}}} \\ &= a^n \end{aligned}$$

In entrambi i casi, i nuovi valori per  $r$ ,  $b$  e  $i$  soddisfano la condizione (I), *invariante del ciclo*. Cosa ne consegue? Quando il ciclo termina, (I) vale nuovamente, quindi:

$$r \cdot b^i = a^n$$

Inoltre, sappiamo che  $i = 0$ , perché il ciclo è terminato.

Dato che  $i = 0$ , si ha che  $r \cdot b^i = r \cdot b^0 = r$ . Ne consegue che  $r = a^n$ , e il metodo calcola effettivamente l'ennesima potenza di  $a$ .

Questa tecnica è piuttosto utile, perché può servire per illustrare un algoritmo che non sia del tutto ovvio. La condizione (I) si chiama invariante del ciclo, perché è vera all'inizio del ciclo, all'inizio di ciascuna iterazione e alla fine del ciclo. Se la condizione invariante di un ciclo viene scelta con competenza, può consentire di dimostrare la correttezza di un calcolo. Consultate *Programming Pearls* (Jon Bentley, Addison-Wesley 1986, Capitolo 4) per un altro esempio interessante.

## 6.6 Usare un debugger

Come avrete certamente ormai capito, raramente i programmi funzionano perfettamente al primo tentativo e spesso scovare gli errori può essere molto frustrante. Naturalmente, potete inserire messaggi di tracciamento per evidenziare il flusso di esecuzione del programma e i valori delle variabili fondamentali, eseguire il programma e tentare di analizzarne il prospetto ottenuto. Se tale analisi non indica chiaramente il problema, potrebbe essere necessario aggiungere o eliminare comandi di stampa ed eseguire nuovamente il programma. È un meccanismo che può richiedere molto tempo.

I moderni ambienti di sviluppo contengono speciali programmi, detti *debugger*, che vi aiutano a localizzare gli errori, permettendovi di seguire l'esecuzione di un programma che state sviluppando. Potete interrompere e far proseguire il vostro programma, per poi vedere il contenuto delle variabili quando arrestate temporaneamente l'esecuzione. A ciascuna pausa, potete scegliere quali variabili esaminare e quanti passi di programma eseguire prima dell'interruzione successiva.

Alcuni credono che i debugger siano solo uno strumento per rendere pigri i programmati. In verità, qualcuno scrive programmi sciatti e poi li sistema alla meglio grazie al debugger, ma la maggioranza tenta onestamente di scrivere il miglior programma possibile, prima di provare a eseguirlo mediante il debugger. Questi programmati sanno che il debugger, sebbene sia più comodo degli enunciati di visualizzazione aggiunti al codice,

**Un debugger è un programma che potete usare per eseguire un altro programma e analizzare il suo comportamento durante l'esecuzione.**

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 vostri programmi, più difficile sarà correggerli inserendo semplicemente enunciati di stampa e scoprirete che il tempo investito per imparare a usare il debugger verrà ampiamente ripagato nella vostra carriera di programmatore.

Come per i compilatori, i debugger differiscono ampiamente da un sistema all'altro. In alcuni sistemi sono alquanto rudimentali e obbligano il programmatore a memorizzare una piccola serie di comandi misteriosi, in altri hanno, invece, un'intuitiva interfaccia a finestre. Le schermate presentate in questo capitolo mostrano 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, contengono un debugger; JSwat è, invece, un debugger a sé stante, gratuito, reperibile all'indirizzo [www.bluemarsh.com/java/jswat](http://bluemarsh.com/java/jswat).

Dovrete scoprire autonomamente come preparare un programma per il *debugging* e come avviare il debugger nel vostro sistema. 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*) 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 scoprirne 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, nel quale l'esecuzione viene sospesa e viene visualizzato, come in Figura 7, 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.

I punti di arresto rimangono attivi finché non vengono esplicitamente rimossi, per cui dovete 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 è diverso per ciascun debugger: alcuni visualizzano sempre una finestra che contiene i valori delle variabili locali; in altri dovete eseguire un comando del tipo “esamina la variabile” e digitare o selezionare il nome della variabile, dopodiché il debugger visualizzerà il contenuto della variabile. Se tutte le variabili contengono quello che vi aspettate, potete eseguire il programma fino al punto successivo in cui volete fermarlo.

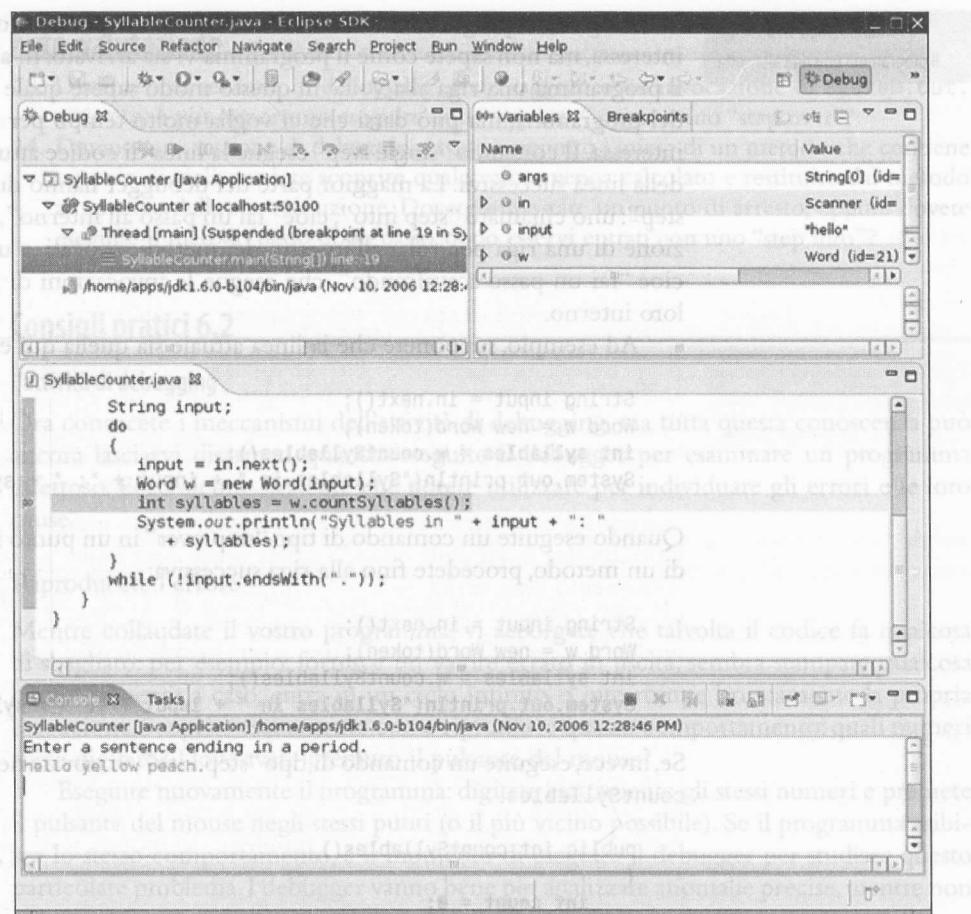
Quando esamineate oggetti, spesso avrete bisogno di fornire un comando per “aprire” l'oggetto, ad esempio selezionando con il mouse un nodo di un albero. Una volta aperto l'oggetto, potrete vedere le sue variabili di esemplare, come in Figura 8.

**Potete usare efficacemente il debugger comprendendo a fondo tre concetti: breakpoint, esecuzione single step e ispezione di variabili.**

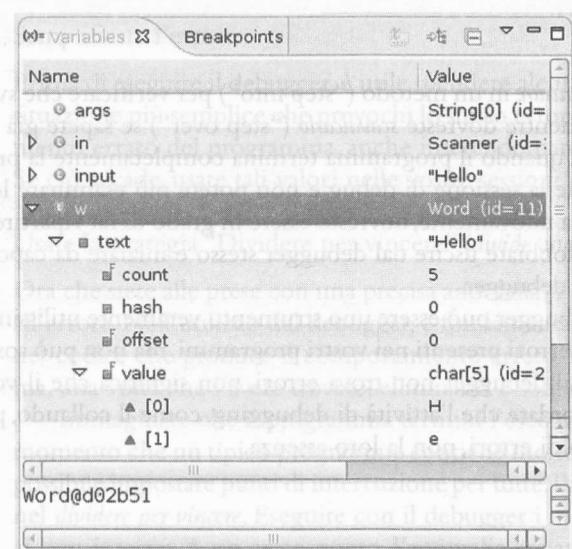
**Quando il debugger esegue un programma, l'esecuzione viene interrotta ogni volta che viene raggiunto un punto di arresto.**

**Figura 7**

Il debugger fermo a un punto di arresto

**Figura 8**

Ispezionare variabili



Il comando *single step* esegue il programma una riga alla volta.

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 linea di codice attuale e si interrompe prima della linea 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(token);
int syllables = w.countSyllables();
System.out.println("Syllables in " + input + ": " + syllables);
```

Quando eseguite un comando di tipo “step over” in un punto in cui esiste l’invocazione di un metodo, procedete fino alla riga successiva:

```
String input = in.next();
Word w = new Word(token);
int syllables = w.countSyllables();
System.out.println("Syllables in " + input + ": " + syllables);
```

Se, invece, eseguite un comando di tipo “step into”, vi trovate 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 e non potete più esaminare le variabili. Per eseguire il programma nuovamente, dovreste essere in grado di far ripartire il debugger, oppure può darsi che dobbiate uscire dal debugger stesso e iniziare da capo: i particolari dipendono dal tipo di debugger.

Un debugger può essere uno strumento veramente utilissimo per individuare ed eliminare gli errori presenti nei vostri programmi, ma non può sostituire una progettazione attenta: se il debugger non trova errori, non significa che il vostro programma non ne abbia. Ricordate che l’attività di debugging, come il collaudo, può soltanto testimoniare la presenza di errori, non la loro assenza.

Il debugger può essere usato soltanto per testimoniare la presenza di errori, non la loro assenza.



## Auto-valutazione

13. Durante una sessione di debugging, avete raggiunto un'invocazione di `System.out.println`: è più opportuno eseguire uno "step into" oppure uno "step over"?
14. 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. Dovete impostare un punto di arresto, oppure dovete eseguire il metodo passo dopo passo, dopo esservi entrati con uno "step into"?



## 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 accorgete che talvolta il codice fa qualcosa di sbagliato: per esempio, 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 nelle vostre sessioni di utilizzo del debugger.

#### Fase 3 Usate la strategia "Dividere per vincere" (*divide and conquer*)

Ora che siete alle prese con una precisa anomalia, è opportuno avvicinarsi il più possibile all'errore. Nell'utilizzo del debugger, 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*. Eseguite con il debugger i metodi invocati dal `main`, senza entrare al loro interno. A un certo punto, l'anomalia apparirà e saprete quale metodo contiene

Per identificare il punto in cui il programma fallisce, usate la tecnica del "dividere per vincere".

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, 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 dovete 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 *fà* il programma, ma voi dovete sapere cosa *dovrebbe fare*, altrimenti non sarete in grado di scoprire gli errori. Prima di tracciare passo per 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 vedere. Se non ne avete idea, prendetevi un po' di tempo e riflettete prima di agire. Munitevi di una pratica calcolatrice tascabile per eseguire calcoli indipendentemente dal programma ed esamineate la variabile quando sapete quale deve essere il suo valore corretto: questo è il momento della verità. Se il programma è ancora sulla strada giusta, il valore sarà quello previsto e potrete cercare l'errore più avanti. Se il valore è diverso, potreste aver trovato qualcosa. Controllate nuovamente i vostri calcoli: se siete sicuri che il valore sia corretto, scoprите perché il programma giunge a una conclusione diversa.

In molti casi, gli errori di un programma sono il risultato di sviste banali, come l'errore per scarto di uno nella condizione per l'uscita da 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. Diversamente dal vostro assistente di matematica, i programmi (come i problemi del mondo reale) non fanno uno sforzo speciale per assicurarsi che tutti i calcoli si svolgano con semplici numeri interi, 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, e quindi esaminando le variabili per verificarlo.

#### Fase 5. Controllate tutti i dettagli

Quando fate il debugging di un programma, spesso avete già un'idea della natura del problema. Nondimeno, mantenete una mentalità aperta e guardate a tutti i particolari circostanti. 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 l'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 grossa tentazione di “mettere un rattoppo”, 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 che esiste un problema importante in merito alla logica del programma: con il debugger si può fare poco, occorre ripensare 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, 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 d’ingresso possono ancora contenere segni di punteggiatura, ma non vogliamo che questi facciano parte della parola.

Ecco il codice sorgente per la classe, contenente un paio di errori.

### File ch06/debugger/Word.java

```
/*
 * Questa classe descrive parole di un documento.
 * La classe contiene un paio di errori.
 */
public class Word
{
    private String text;

    /**
     * Costruisce una parola eliminando caratteri iniziali e finali
     * che non siano lettere, come i segni di punteggiatura.
     * @param s la stringa di ingresso
     */
    public Word(String s)
    {
        int i = 0;
```

```

        while (i < s.length() && !Character.isLetter(s.charAt(i)))
            i++;

        int j = s.length() - 1;
        while (j > i && !Character.isLetter(s.charAt(j)))
            j--;
        text = s.substring(i, j);
    }

    /**
     * Restituisce il testo della parola.
     * @return il testo della parola
    */
    public String getText()
    {
        return text;
    }

    /**
     * Conta le sillabe nella parola.
     * @return il numero di sillabe
    */
    public int countSyllables()
    {
        int count = 0;
        int end = text.length() - 1;
        if (end < 0) return 0; // la stringa vuota non ha sillabe

        // una 'e' alla fine della parola non conta come vocale
        char ch = Character.toLowerCase(text.charAt(end));
        if (ch == 'e') end--; // riga 47 del codice (per il debugging)

        boolean insideVowelGroup = false;
        for (int i = 0; i <= end; i++)
        {
            ch = Character.toLowerCase(text.charAt(i));
            String vowels = "aeiouy";
            if (vowels.indexOf(ch) >= 0)
            {
                // ch è una vocale
                if (!insideVowelGroup)
                {
                    // inizia un nuovo gruppo di vocali
                    count++;
                    insideVowelGroup = true;
                }
            }
        }

        // ogni parola ha almeno una sillaba
        if (count == 0)
            count = 1;
    }

    return count;
}
}

```

Ecco una semplice classe di prova. Digitate una frase e verrà visualizzato il conteggio delle sillabe di tutte le sue parole.

### File ch06/debugger/SyllableCounter.java

```
import java.util.Scanner;

/**
 * Questo programma conta le sillabe di tutte le parole di una frase.
 */

public class SyllableCounter
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);

        System.out.println("Enter a sentence ending in a period.");

        String input;
        do
        {
            input = in.next();
            Word w = new Word(input);
            int syllables = w.countSyllables();
            System.out.println("Syllables in " + input
                + ": " + syllables);
        }
        while (!input.endsWith("."));
    }
}
```

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.

Come prima cosa, 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 47 (osservate la Figura 9).

Ora ispezionate la variabile `ch`. Come potete vedere in Figura 10, questo particolare debugger ha un comodo visualizzatore per le variabili locali e di esemplare; se il vostro non lo ha, può darsi che dobbiate ispezionare `ch` manualmente. Potete verificare che `ch`

**Figura 9**

Attività di debugging  
nel metodo countSyllables

```

    /*
     * @return the syllable count
     */
    public int countSyllables()
    {
        int count = 0;
        int end = text.length() - 1;
        if (end < 0) return 0; // The empty string has no syllables

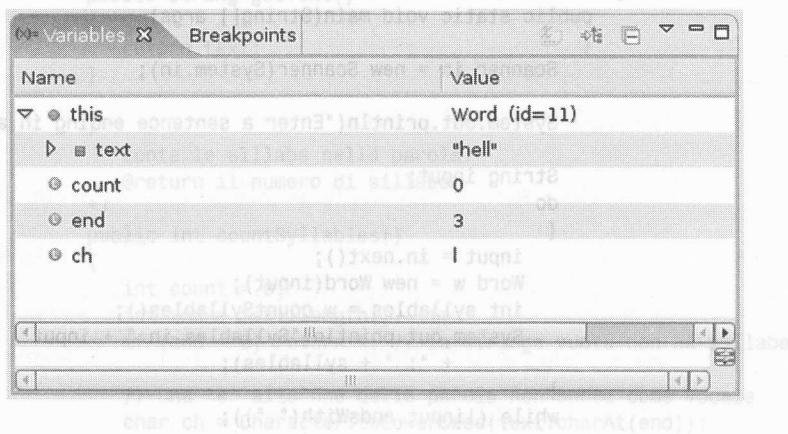
        // An e at the end of the word doesn't count as a vowel
        char ch = Character.toLowerCase(text.charAt(end));
        if (ch == 'e') end--;

        boolean insideVowelGroup = false;
    }
}

```

**Figura 10**

Valori delle variabili locali  
e di esemplare



contiene il valore '1', cosa un po' strana. Osservate il codice sorgente: la variabile `end` è stata impostata al valore `text.length() - 1`, l'ultima posizione nella stringa `text`, e `ch` è il carattere che si trova in tale posizione.

Proseguendo nell'analisi, troverete 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`.

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. Fornite 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 in Figura 11, impostate un breakpoint dopo la fine del secondo ciclo, in modo da poter ispezionare i valori di `i` e `j`.

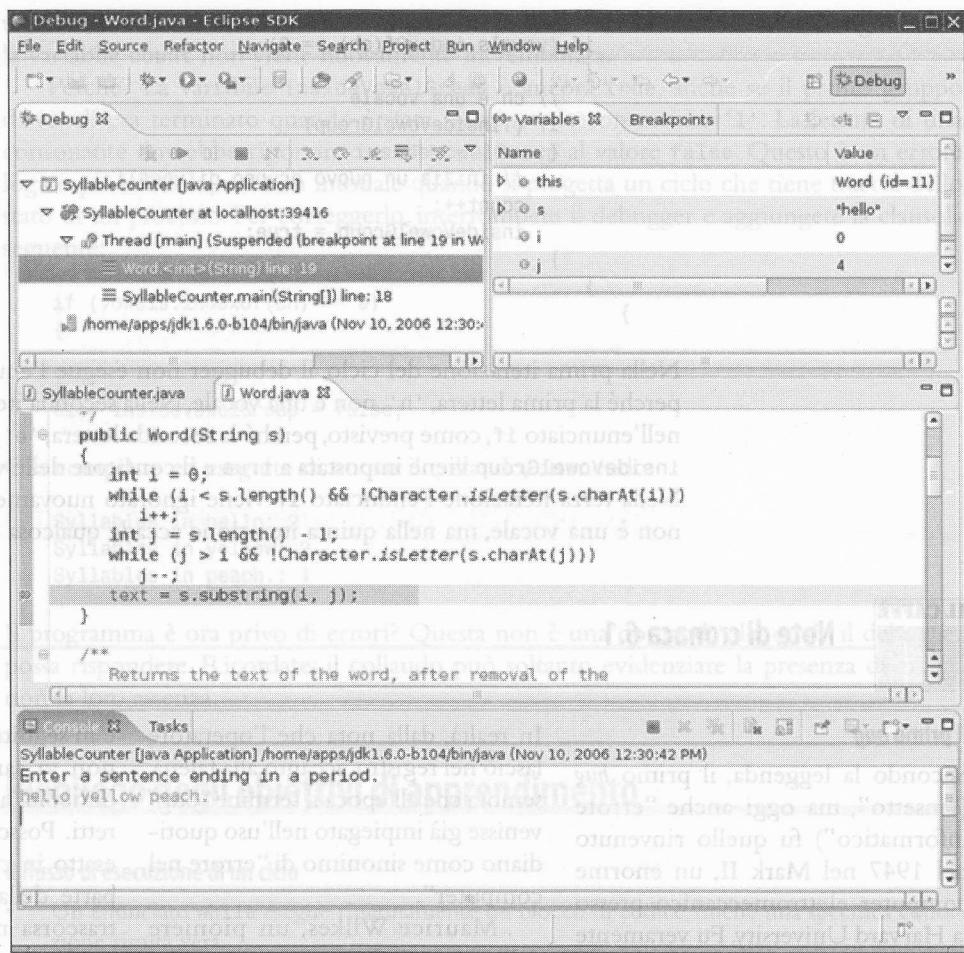
A questo punto, l'ispezione di `i` e `j` mostra che `i` vale 0 e che `j` vale 4. Ciò ha senso, perché non ci sono segni di punteggiatura da ignorare. Quindi, perché `text` viene impostata a "hell"? Ricordate che il metodo `substring` considera le posizioni fino al proprio secondo parametro *senza includerlo*, per cui l'invocazione corretta dovrebbe essere

`text = s.substring(i, j + 1);`

Questo è un tipico errore "per scarto di uno".

**Figura 11**

Attività di debugging nel costruttore di Word



Correggete questo errore, compilate il programma e collaudate di nuovo il programma nei tre casi di prova. Otterrete la seguente visualizzazione in uscita:

```
Syllables in hello: 1
Syllables in yellow: 1
Syllables in peach.: 1
```

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 = Character.toLowerCase(text.charAt(i));
```

```
String vowels = "aeiouy";
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 ignorato nuovamente, perché la lettera '`l`' non è una vocale, ma nella quinta iterazione accade qualcosa di strano. La lettera '`o`' è



## Note di cronaca 6.1

## Il primo *bug*

Secondo la leggenda, il primo *bug* (“insetto”, ma oggi anche “errore informatico”) fu quello rinvenuto nel 1947 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 ricordarmi il momento esatto in cui mi fu chiaro che gran parte della mia vita futura sarebbe trascorsa nel trovare gli errori nei miei programmi”.

### Il primo *bug*

1100 Started Cosine Tape (Sine chart)  
1525 ~~Started~~ <sup>70</sup> ~~cross~~  
1545 Relay #70 Panel F  
(Moth) in relay.  
  
First actual case of bug being found.  
1600 ~~1600~~ instant start.  
1700 closed form.

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 un processo. 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.

## Riepilogo degli obiettivi di apprendimento

### Il flusso di esecuzione di un ciclo

- Un enunciato `while` esegue ripetutamente un blocco di codice, finché 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.

### 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.
- Nei cicli, valutate l'esigenza di limiti simmetrici o asimmetrici.
- Contate il numero di iterazioni di un ciclo, per verificare che sia corretto.

### Un ciclo può elaborare dati fino all'arrivo di un valore sentinella

- A volte la condizione di terminazione di un ciclo può essere valutata soltanto all'interno del ciclo stesso, che può essere controllato mediante una variabile booleana.

### Con cicli annidati si realizzano più livelli di iterazione

- I cicli possono essere annidati. Un esempio tipico di ciclo annidato è quello che visualizza tabelle con righe e colonne.

### Le simulazioni con numeri casuali usano cicli

- In una simulazione si generano ripetutamente numeri casuali, usandoli per simulare una determinata attività.

### Con un debugger si individuano errori durante l'esecuzione

- Un *debugger* è un programma che potete usare per eseguire un altro programma e analizzare il suo comportamento durante l'esecuzione.
- Potete usare efficacemente il debugger comprendendo a fondo tre concetti: *breakpoint*, esecuzione *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.
- Il debugger può essere usato soltanto per testimoniare la presenza di errori, non la loro assenza.
- 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.

## Classi, oggetti e metodi presentati nel capitolo

```
java.util.Random
    nextDouble
    nextInt
```

## Esercizi di ripasso

- \*\* Esercizio R6.1.** Quali enunciati di ciclo esistono in Java? Elencate semplici regole che descrivano l'utilizzo tipico di ciascun tipo di ciclo.
- \*\* Esercizio R6.2.** Che cosa stampa il codice seguente?

```
for (int i = 0; i < 10; i++)
{
    for (int j = 0; j < 10; j++)
        System.out.print(i * j % 10);
    System.out.println();
}
```

- \*\* Esercizio R6.3.** Quante ripetizioni eseguono i cicli seguenti, se *i* non viene modificato nel corpo del ciclo ed è una variabile di tipo intero?

- a. `for (i = 1; i <= 10; i++) ...`
- b. `for (i = 0; i < 10; i++) ...`
- c. `for (i = 10; i > 0; i--) ...`
- d. `for (i = -10; i <= 10; i++) ...`
- e. `for (i = 10; i >= 0; i++) ...`
- f. `for (i = -10; i <= 10; i = i + 2) ...`
- g. `for (i = -10; i <= 10; i = i + 3) ...`

- \* Esercizio R6.4.** Riscrivete il codice seguente, sostituendo il ciclo `for` con un ciclo `while`.

```
int s = 0;
for (int i = 1; i <= 10; i++) s = s + i;
```

- \*\* **Esercizio R6.5.** Riscrivete il codice seguente, sostituendo il ciclo `do` con un ciclo `while`.

```
int n = 1;
double x = 0;
double s;
do
{
    s = 1.0 / (n * n);
    x = x + s;
    n++;
}
while (s > 0.01);
```

- \* **Esercizio R6.6.** Cos'è un ciclo infinito? Nel vostro computer, in che modo potete terminare un programma che sta eseguendo un ciclo infinito?
- \*\*\* **Esercizio R6.7.** Delineate tre diverse strategie per realizzare il seguente "ciclo e mezzo":

Ripeti

Leggi il nome del ponte.  
Se non è valido, esci dal ciclo.  
Leggi la lunghezza del ponte, misurata in piedi.  
Se non è valida, esci dal ciclo.  
Converti la lunghezza in metri.  
Visualizza i dati relativi al ponte.

Utilizzate una variabile booleana oppure un enunciato `break` oppure, ancora, un metodo con più enunciati `return`. Quale di questi tre approcci vi sembra più chiaro?

- \* **Esercizio R6.8.** Realizzate un ciclo che chiede all'utente di inserire un numero compreso tra 1 e 10, consentendo tre tentativi.
- \* **Esercizio R6.9.** A volte gli studenti scrivono programmi con istruzioni del tipo "Inserisci un dato, zero per terminare", terminando l'inserimento dei dati in ingresso quando l'utente inserisce il numero zero. Spiegate perché questa non è una buona idea.
- \* **Esercizio R6.10.** In che modo si può usare un generatore di numeri casuali per simulare l'estrazione di una carta da gioco?
- \* **Esercizio R6.11.** Cos'è l'errore "per scarto di uno"? Fornite un esempio tratto dalla vostra esperienza di programmazione.
- \*\* **Esercizio R6.12.** 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.
- \* **Esercizio R6.13.** Cosa sono i cicli annidati? Fornite un esempio in cui solitamente si usano cicli annidati.
- \*T **Esercizio R6.14.** 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")
- \*\*T **Esercizio R6.15.** Spiegate in dettaglio come esaminare, con il vostro debugger, le informazioni contenute in un oggetto di tipo `String`.
- \*\*T **Esercizio R6.16.** Spiegate in dettaglio come esaminare, con il vostro debugger, le informazioni contenute in un oggetto di tipo `Rectangle`.

\*\*T **Esercizio R6.17.** Spiegate in dettaglio come usare il vostro debugger per esaminare il saldo contenuto in un oggetto di tipo `BankAccount`.

\*\*T **Esercizio R6.18.** Spiegate la strategia “dividere per vincere”, utilizzata per avvicinarsi a un errore usando il debugger.

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

# 7

## Vettori e array

### Obiettivi del capitolo

- Acquisire familiarità con l'utilizzo di array e vettori
- Studiare le classi involucro, la tecnica di *auto-boxing* e il ciclo **for** esteso
- Apprendere gli algoritmi più comuni per gli array
- Capire come usare array bidimensionali
- Imparare a scegliere tra array e vettori
- Realizzare array riempiti solo in parte
- Apprendere il concetto di collaudo regressivo

Per poter elaborare grandi quantità di dati, c'è bisogno di memorizzarle in strutture apposite: quelle più comunemente utilizzate in Java sono gli array e i vettori (o "liste ad accesso casuale", *array list*). In questo capitolo imparerete a costruire array e vettori, a memorizzarvi valori e ad accedere a tali valori. Presentiamo qui anche il ciclo **for** esteso, un comodo enunciato per l'elaborazione di tutti gli elementi presenti in una di queste strutture: vedrete come realizzare i più comuni algoritmi di elaborazione di dati contenuti in array usando il ciclo **for** esteso e i cicli ordinari. Il capitolo si conclude con una presentazione degli array bidimensionali, utili per la gestione di dati organizzati in righe e colonne.

## 7.1 Array

**Un array è una sequenza di valori del medesimo tipo, cioè omogenei.**

In molti programmi c'è bisogno di manipolare insiemi di dati fra loro correlati e sarebbe assai scomodo utilizzare sequenze di variabili del tipo `value1`, `value2`, `value3`, ... e così via. La struttura denominata *array* ("schiera") mette a disposizione uno strumento migliore per memorizzare un insieme di valori.

Un *array* è una sequenza di valori del medesimo tipo, cioè *omogenei*, e i valori memorizzati al suo interno vengono chiamati *elementi*. Ad esempio, ecco come si costruisce un array contenente dieci valori in virgola mobile:

```
new double[10]
```

Il numero di elementi (10, in questo caso) costituisce la *lunghezza* dell'array.

L'operatore `new` costruisce semplicemente l'array: memorizzerete in una variabile il riferimento all'array, in modo da accedervi successivamente.

Il tipo di una "variabile array" (cioè di una "variabile riferimento che consente di accedere a un oggetto di tipo array") è il tipo degli elementi dell'array, seguito da `[]`: in questo esempio il tipo è `double[]`, perché gli elementi sono di tipo `double`. Ecco la dichiarazione di una variabile array:

```
double[] values = new double[10];
```

Quindi, `values` è un riferimento a un array di numeri in virgola mobile: tale riferimento viene inizializzato in modo da puntare a un array contenente 10 numeri, come si può vedere nella Figura 1.

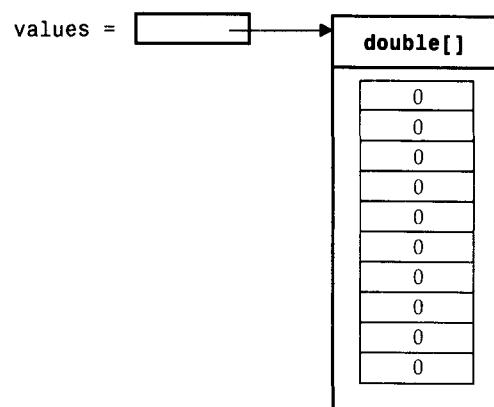
Si possono anche creare array di oggetti, come in questo esempio:

```
BankAccount[] accounts = new BankAccount[10];
```

Nel momento in cui un array viene creato, tutti i suoi elementi vengono inizializzati al valore 0 (per un array di numeri come `int[]` o `double[]`), `false` (per un array `boolean[]`) o `null` (per un array di riferimenti a oggetti).

In alternativa, potete inizializzare un array con valori diversi, elencandoli fra una coppia di parentesi graffe, separati da virgolette:

**Figura 1**  
Un riferimento ad array  
e un array



```
int[] primes = { 2, 3, 5, 7, 11 };
```

Il compilatore Java conta gli elementi che volete inserire nell'array, crea un array della dimensione giusta e lo riempie ordinatamente con gli elementi specificati.

Ciascun elemento di un array è identificato da un indice di tipo intero, che viene inserito all'interno delle parentesi quadre. Ad esempio, l'espressione

```
values[4]
```

identifica l'elemento di indice 4 nell'array **values**.

Potete scrivere un valore nella posizione desiderata usando un enunciato di assegnazione, in questo modo:

```
values[2] = 29.95;
```

Si accede a un elemento di un array mediante un indice intero, usando l'operatore `[ ]`.

Ora la posizione di indice 2 in **values** contiene il valore 29.95, come potete vedere nella Figura 2.

Per leggere l'elemento che si trova nella posizione di indice 2, usate semplicemente l'espressione **values[2]** come fareste con qualsiasi variabile di tipo **double**:

```
System.out.println("The price of this item is " + values[2]);
```

Se guardate con attenzione alla Figura 2, noterete che i valori degli indici iniziano da 0, cioè

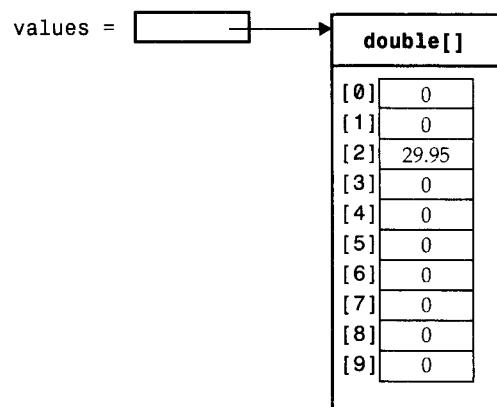
**values[0]** è il primo elemento  
**values[1]** è il secondo elemento  
**values[2]** è il terzo elemento

In un array, i valori degli indici vanno da zero alla sua lunghezza, diminuita di uno.

e così via. Questa convenzione può sembrare strana a chi si avvicina per la prima volta alla programmazione, per cui dovrete porre particolare cura ai valori degli indici. In particolare, l'*ultimo* elemento di un array è identificato da un indice uguale alla lunghezza dell'array *meno uno*. Ad esempio, **values** si riferisce a un array di lunghezza 10, per cui il suo ultimo elemento è **values[9]**.

**Figura 2**

Modifica di un elemento in un array



L'accesso a un elemento inesistente costituisce un errore di limiti.

L'espressione `array.length` ha sempre un valore uguale al numero di elementi dell'array.

Il tentativo di accedere a una posizione inesistente provoca il lancio di un'eccezione di tipo “indice di array al di fuori dei limiti” (*array index out of bounds*), come nell'esempio seguente, perché `values[10]` non esiste:

```
values[10] = 29.95; // ERRORE DI LIMITI
```

Per evitare questo tipo di errori, vi servirà conoscere il numero di elementi che costituiscono 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: si tratta di una variabile di esemplare dell'oggetto `array`, non di un metodo. Tuttavia, a questa variabile di esemplare non è possibile assegnare un nuovo valore: in altre parole, `length` è una variabile di esemplare `final public`. Una bella anomalia, dato che, normalmente, i programmati Java usano un metodo per ispezionare le proprietà di un oggetto: in ogni caso, dovete solo ricordarvi di omettere le parentesi.

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 7.6.

**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 NUMBERS_LENGTH = 10;</code><br><code>int[] numbers = new int[NUMBERS_LENGTH];</code>      | Invece di un “numero magico”, è bene usare una costante.                                                                |
| <code>int valuesLength = in.nextInt();</code><br><code>double[] values = new double[valuesLength];</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[] names = new String[3];</code>                                                              | Un array di tre riferimenti a stringa, tutti inizialmente al valore <code>null</code> .                                 |
| <code>String[] fr = { "Emily", "Bob", "Cindy" };</code>                                                   | Un altro 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> . |



## Auto-valutazione

1. Quali valori sono presenti nell'array `values` dopo l'esecuzione di questi enunciati?

```
double[] values = new double[10];
for (int i = 0; i < values.length; i++) values[i] = i * i;
```

2. Cosa visualizzano i seguenti frammenti di programma? Oppure, se c'è un errore, descrivetelo e specificate se si tratta di un errore di compilazione o di un errore di esecuzione.  
a. `double[] a = new double[10];`

```

    System.out.println(a[0]);
b. double[] b = new double[10];
    System.out.println(b[10]);
c. double[] c;
    System.out.println(c[0]);

```

## Errori comuni 7.1

### Errori di limiti

Nell'uso di array, l'errore più frequente è l'accesso a una posizione che non esiste.

```

double[] data = new double[10];
data[10] = 29.95;
// Errore: data ha soltanto gli elementi con indice compreso tra 0 e 9

```

Quando il programma viene eseguito, l'uso di un indice fuori dai limiti genera un'eccezione e arresta il programma stesso.

Si tratta di un notevole passo avanti rispetto a linguaggi come C e C++, nei quali non viene prodotto alcun messaggio di errore e il programma si limita ad alterare senza problemi (o, forse, con qualche problema...) la cella di memoria che si trova a 10 posizioni di distanza dalla cella iniziale dell'array. A volte il problema può passare inosservato, altre volte il programma si comporterà in modo bizzarro o morirà di una morte orribile parecchie istruzioni più tardi. Sono errori gravi, che possono essere difficili da individuare in un programma scritto in C o C++ e che, in tali linguaggi, costituiscono

## Sintassi di Java

### 7.1 Array

#### Sintassi

Per costruire un array: `new nomeTipo[lunghezza]`

Per accedere a un elemento: `riferimentoAdArray[indice]`

#### Esempio

| Nome della variabile array              | /                                              | Lunghezza | Elementi inizializzati<br>a zero. |
|-----------------------------------------|------------------------------------------------|-----------|-----------------------------------|
| <code>Type della variabile array</code> | <code>double[] values = new double[10];</code> |           |                                   |

`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.

una delle cause principali di vulnerabilità per la sicurezza, come descritto in Note di cronaca 7.1.



## Errori comuni 7.2

### Array non inizializzati o non riempiti

Un altro errore frequente consiste nel dichiarare un riferimento ad array, senza assegnargli un array reale.

```
double[] values;
values[0] = 29.95; // Errore: values non è inizializzato
```

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];
// contiene dieci riferimenti null
```

Questo array contiene dieci riferimenti `null`, non dieci conti bancari nel loro stato predefinito. Dovete ricordarvi di riempire l'array, ad esempio in questo modo:

```
for (int i = 0; i < 10; i++)
{
    accounts[i] = new BankAccount();
}
```



## Consigli per la qualità 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

```
double[] personalData = new double[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.

```
int age;
double bankBalance;
double shoeSize;
```



## Consigli per la qualità 7.2

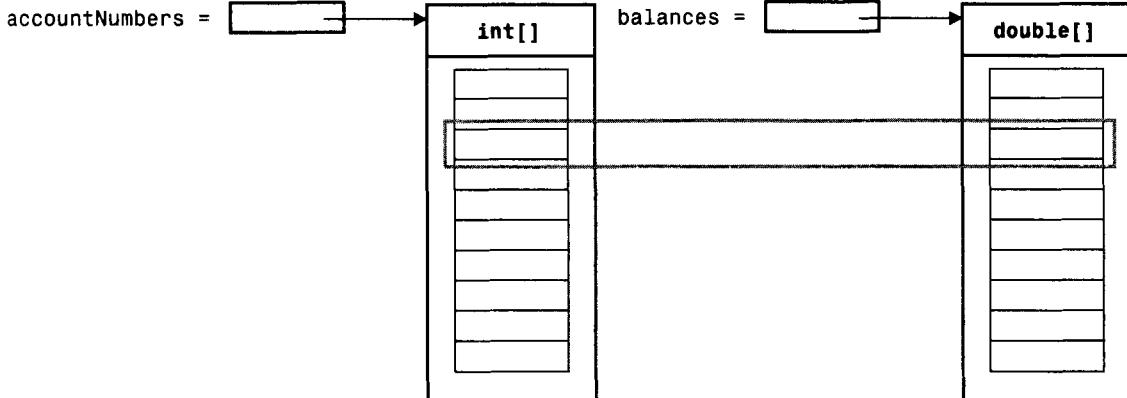
### Trasformare array paralleli in array di oggetti

Ai programmatore 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:

**Figura 3**

Evitate gli array paralleli

```
// non fate così
int[] accountNumbers;
double[] balances;
```



Evitate di usare array paralleli,  
trasformandoli in array di oggetti.

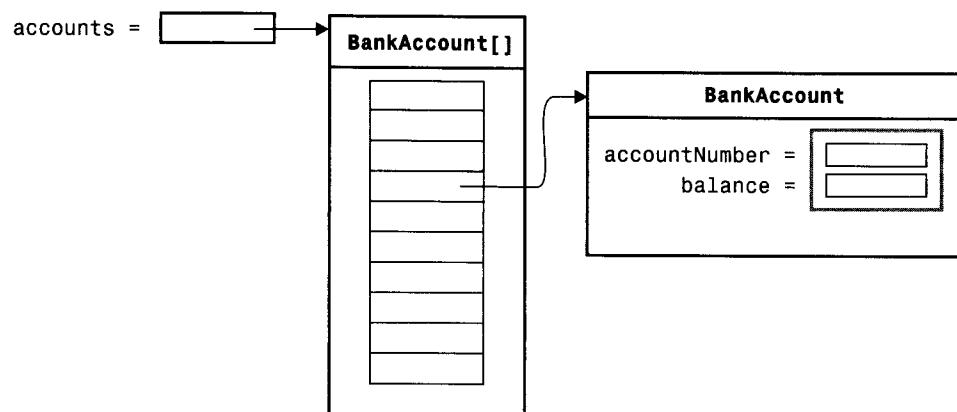
Questi array vengono detti *array paralleli* (osservate la Figura 3), perché la *fetta*  $i$ -esima della struttura contiene dati che devono essere elaborati congiuntamente (`accountNumbers[i]` e `balances[i]`).

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 "fetta" e identificate il concetto che rappresenta: quindi, inserite tale concetto in una classe. Nel nostro esempio ciascuna fetta 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 4. Quali sono i vantaggi? Pensate al futuro. Forse il vostro programma verrà modificato e avrete bisogno di memorizzare anche il proprietario del conto bancario. 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 4**  
Riorganizzate array paralleli  
in array di oggetti



## Argomenti avanzati 7.1

### Metodi con un numero di parametri variabile

A partire dalla versione 5 di Java, è possibile dichiarare metodi che ricevono un numero di parametri variabile. Ad esempio, si può modificare il metodo `add` della classe `DataSet` vista nel Capitolo 6 in modo che possano essere aggiunti più valori con un'unica invocazione:

```
data.add(1, 3, 7);
data.add(4);
data.add(); // corretto ma inutile
```

Tale metodo `add` modificato va dichiarato in questo modo:

```
public void add(double... values)
```

Il simbolo `...` indica proprio che il metodo può ricevere un numero qualsiasi di valori di tipo `double`. Il parametro `values` è, in realtà, un array di tipo `double[]`, che contiene 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 class DataSet
{
    ...
    public void add(double... values)
    {
        for (int i = 0; i < values.length; i++) // values è un double[]
        {
            double x = values[i];
            sum = sum + x;
            if (count == 0 || maximum < x) maximum = x;
            count++;
        }
    }
}
```



## Note di cronaca 7.1

### Uno dei primi worm di Internet

Nel novembre del 1988 uno studente della Cornell University lanciò un programma con un virus che infettò circa 6000 computer collegati a Internet in tutti gli Stati Uniti. Decine di migliaia di utenti di computer non furono più in grado di leggere i propri messaggi di posta elettronica né di usare il computer in alcun modo. Caddero nella trappola tutte le università più importanti e molte industrie a tecnologia avanzata (Internet era molto più piccola di ora).

Il tipo particolare di virus utilizzato in quella offensiva è detto *worm* ("verme"): il programma del virus "striscia" da un computer all'altro attraverso Internet. Il programma intero era piuttosto complesso, ma qui ci interessa uno dei metodi usati in quell'attacco. Il virus cercava di connettersi a *finger*, un programma del sistema operativo UNIX che serve a reperire informazioni su un utente che abbia accesso a un particolare computer sulla rete. Come molti programmi di UNIX, *finger* era stato scritto in C e quando costruì un array, in C come in Java, dovete pensare a quanti elementi vi servono. Per memorizzare il nome di un utente da cercare (ad esempio, *walters@cs.sjsu.edu*) il programma *finger* usava un array di 512 caratteri, nell'ipotesi che nessuno avrebbe mai fornito un nome di utente tanto lungo. Sfortunatamente, il linguaggio C, a differenza di Java, non verifica che l'indice di un array sia minore della lunghezza dell'array: se scrivete in un array usando un indice troppo grande, sovrascrivete semplicemente

posizioni di memoria appartenenti ad altri oggetti.

In alcune versioni del programma *finger* il programmatore era stato pigro e non si era preoccupato di verificare se l'array che doveva memorizzare i caratteri forniti in ingresso fosse grande abbastanza per contenerli. Il programma del virus riempiva l'array da 512 caratteri con 536 byte: i 24 byte in eccesso avrebbero sovrascritto un indirizzo di ritorno, che l'attaccante sapeva essere memorizzato subito dopo lo spazio destinato ai dati in ingresso. Quando quella funzione terminava, il controllo dell'esecuzione non ritornava al suo chiamante, ma al codice fornito dal virus (come si può vedere nella figura). Quel codice veniva, quindi, eseguito con gli stessi privilegi del super-utente del sistema operativo, concessi a *finger*, permettendo al virus di ottenere l'ingresso nel sistema remoto.

Se il programmatore che aveva scritto *finger* fosse stato più cosciente, questo particolare attacco non sarebbe stato possibile: tutti i programmatori che usano C e C++ devono stare particolarmente attenti a non superare i limiti degli array, mentre la

macchina virtuale di Java gestisce questa protezione in modo automatico.

Ci si potrebbe chiedere per quale motivo un programmatore esperto passi molte settimane o mesi a pianificare l'atto antisociale di invadere migliaia di computer e di renderli inutilizzabili. Sembra che intenzione dell'autore fosse la semplice invasione, mentre la messa fuori servizio dei computer fosse un effetto secondario dovuto a un errore nella progettazione del virus stesso. L'autore venne condannato a 3 anni di affidamento in prova, 400 ore di servizi sociali e una multa di dieci mila dollari.

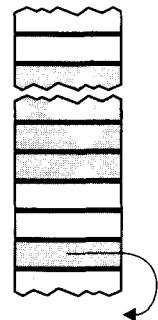
Di recente gli attacchi a computer si sono intensificati e le motivazioni sono diventate più criminali: invece di mettere fuori uso i computer, spesso i virus carpiranno informazioni finanziarie o utilizzano i computer attaccati per inviare posta elettronica con pubblicità indesiderata (*spam*). La cosa più triste è che molti di questi attacchi continuano a essere resi possibili dalla scarsa attenzione dei programmatore, che scrivono ancora programmi vulnerabili da fenomeni di *buffer overrun*.

Un attacco  
di tipo "Buffer  
Overrun"

① Prima dell'attacco

Buffer di riga  
(512 byte)

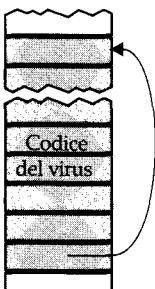
Indirizzo  
di ritorno



② Dopo l'attacco

Buffer  
sovrautilizzato  
(536 byte)

Indirizzo  
di ritorno



## 7.2 Vettori (liste ad accesso casuale)

La classe `ArrayList` gestisce una sequenza di oggetti; la lunghezza della sequenza è variabile.

Gli array sono strutture piuttosto rudimentali. In questo paragrafo presentiamo, invece, la classe `ArrayList` (“vettore” o “lista ad accesso casuale”) che consente la memorizzazione di una raccolta di oggetti, esattamente come un array, ma con 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.

In questo modo si dichiara un vettore di stringhe:

```
ArrayList<String> names = new ArrayList<String>();
```

La classe `ArrayList` è una classe generica:  
`ArrayList<NomeTipo>` contiene oggetti di tipo `NomeTipo`.

Il tipo `ArrayList<String>` specifica un vettore di stringhe. Le parentesi angolari attorno al tipo `String` indicano che `String` è un *tipo (che funge da) parametro*: si può sostituire `String` con qualsiasi altra classe, ottenendo un vettore di tipo diverso, e per questo motivo `ArrayList` viene detta *classe generica*. Nel Capitolo 16 studierete le classi generiche, per il momento usate semplicemente un esemplare di tipo `ArrayList<T>` ogni volta che volete memorizzare raccolte di oggetti di tipo `T`. Ricordate, però, che i tipi primitivi non possono essere utilizzati come tipo parametro: non esistono esemplari di `ArrayList<int>` o di `ArrayList<double>`, ma nel Paragrafo 7.3 vedrete come risolvere il problema.

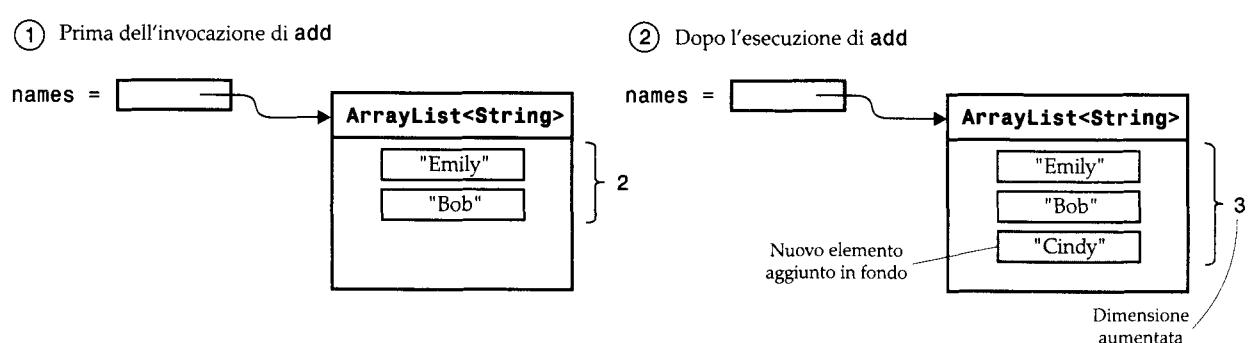
La dimensione iniziale di un vettore, dopo la sua costruzione, è zero: dovete usare il metodo `add` per aggiungere un oggetto alla fine della sequenza di oggetti già contenuti nel vettore, la cui dimensione aumenta a ogni invocazione di `add` (Figura 5); il metodo `size` restituisce la dimensione attuale del vettore.

```
names.add("Emily"); // ora names ha dimensione 1 e l'unico elemento "Emily"
names.add("Bob");   // ora names ha dimensione 2 e elementi: "Emily", "Bob"
names.add("Cindy"); // ora names ha dimensione 3 e elementi: "Emily", "Bob",
                   // "Cindy"
```

Per ispezionare il valore di 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);
```

**Figura 5**  
Aggiunta di un elemento mediante `add`



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 alla posizione 2 del vettore `names` un riferimento alla stringa `"Carolyn"`, sovrascrivendo qualunque valore memorizzato precedentemente.

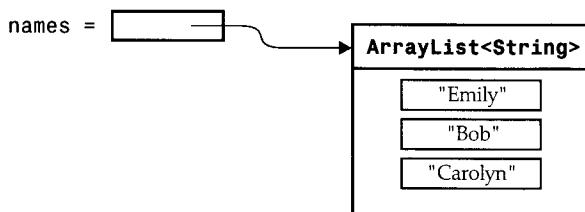
Il metodo `set` può sovrascrivere solamente elementi che già esistono nel vettore ed è diverso dal metodo `add`, che aggiunge un nuovo elemento alla fine del vettore.

È anche possibile inserire un oggetto in una posizione intermedia all'interno di un vettore. L'invocazione `names.add(1, "Ann")` sposta tutti gli elementi di una posizione in avanti, a partire dall'elemento attualmente in posizione 1 fino all'ultimo elemento presente nel vettore, e aggiunge la stringa `"Ann"` nella posizione 1 (Figura 6). Dopo

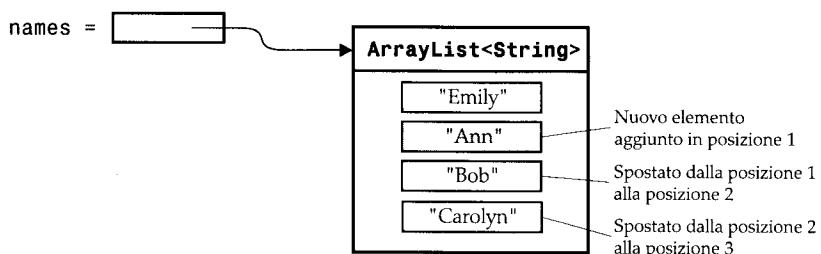
**Figura 6**

Aggiunta e rimozione di un elemento in una posizione intermedia di un vettore

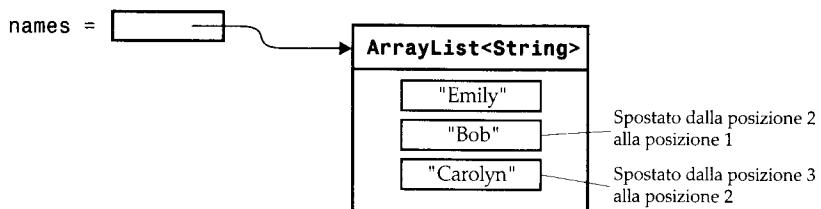
① Prima dell'invocazione di `add`



② Dopo l'esecuzione di `names.add(1, "Ann")`



③ Dopo l'esecuzione di `names.remove(1)`



ogni invocazione del metodo `add` la dimensione del vettore risulta essere aumentata di uno.

Al contrario, il metodo `remove` elimina l'elemento che si trova in una determinata posizione, sposta di una posizione all'indietro tutti gli elementi che si trovano dopo l'elemento rimosso e diminuisce di uno la dimensione del vettore (ultima parte della Figura 6).

**Tabella 2**  
Vettori al lavoro

|                                                                                                                                                                                                         |                                                                                                |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------|
| <code>ArrayList&lt;String&gt; names = new ArrayList&lt;String&gt;();</code>                                                                                                                             | Costruisce un vettore vuoto, adatto a contenere stringhe.                                      |
| <code>names.add("Ann"); names.add("Cindy");</code>                                                                                                                                                      | Aggiunge elementi alla fine.                                                                   |
| <code>System.out.print(names);</code>                                                                                                                                                                   | Visualizza [Ann, Cindy].                                                                       |
| <code>names.add(1, "Bob");</code>                                                                                                                                                                       | Inserisce un elemento in posizione 1, dopodiché <code>names</code> contiene [Ann, Bob, Cindy]. |
| <code>names.remove(0);</code>                                                                                                                                                                           | Elimina l'elemento in posizione 0, dopodiché <code>names</code> contiene [Bob, Cindy].         |
| <code>names.set(0, "Bill");</code>                                                                                                                                                                      | Sostituisce un elemento, dopodiché <code>names</code> contiene [Bill, Cindy].                  |
| <code>String name = names.get(i);</code>                                                                                                                                                                | Ispeziona un elemento.                                                                         |
| <code>String last = names.get(names.size() - 1);</code>                                                                                                                                                 | Ispeziona l'ultimo elemento.                                                                   |
| <code>ArrayList&lt;Integer&gt; squares = new ArrayList&lt;Integer&gt;();</code><br><code>for (int i = 0; i &lt; 10; i++)</code><br><code>{</code><br><code>squares.add(i * i);</code><br><code>}</code> | Costruisce un vettore che contiene i quadrati dei primi dieci numeri interi.                   |

## Sintassi di Java

## 7.2 Vettori

### Sintassi

Per costruire un vettore:

`new ArrayList<nomeTipo>()`

Per accedere a un elemento:

`riferimentoAVettore.get(indice)`  
`riferimentoAVettore.set(indice, valore)`

### Esempio

Tipo della variabile      Nome della variabile      Un vettore di dimensione zero

`ArrayList<String> friends = new ArrayList<String>();`

Per accedere  
a un elemento  
si usano i metodi  
`get` e `set`.

`friends.add("Cindy");`  
`String name = friends.get(i);`  
`friends.set(i, "Harry");`

Il metodo `add` aggiunge  
un elemento in fondo al vettore,  
aumentandone la dimensione.

L'indice deve essere  $\geq 0$   
e  $< \text{friends.size}()$ .

Il programma seguente illustra l'utilizzo della classe `ArrayList` per memorizzare un insieme di oggetti di tipo `BankAccount`, in una versione della classe migliorata rispetto a quella vista nel Capitolo 3: ora ogni conto bancario ha un numero di conto. Notate che viene importata la classe generica `java.util.ArrayList`, senza il tipo di parametro.

### File ch07/arraylist/ArrayListTester.java

```
import java.util.ArrayList;

/**
 * Questo programma collauda la classe ArrayList.
 */
public class ArrayListTester
{
    public static void main(String[] args)
    {
        ArrayList<BankAccount> accounts
            = new ArrayList<BankAccount>();
        accounts.add(new BankAccount(1001));
        accounts.add(new BankAccount(1015));
        accounts.add(new BankAccount(1729));
        accounts.add(1, new BankAccount(1008));
        accounts.remove(0);

        System.out.println("Size: " + accounts.size());
        System.out.println("Expected: 3");
        BankAccount first = accounts.get(0);
        System.out.println("First account number: "
            + first.getAccountNumber());
        System.out.println("Expected: 1008");
        BankAccount last = accounts.get(accounts.size() - 1);
        System.out.println("Last account number: "
            + last.getAccountNumber());
        System.out.println("Expected: 1729");
    }
}
```

### File ch07/arraylist/BankAccount.java

```
/**
 * Un conto bancario ha un saldo che può essere
 * modificato da depositi e prelievi.
 */
public class BankAccount
{
    private int accountNumber;
    private double balance;

    /**
     * Costruisce un conto bancario con saldo uguale a zero.
     * @param anAccountNumber il numero di questo conto bancario
     */
    public BankAccount(int anAccountNumber)
    {
```

```
        accountNumber = anAccountNumber;
        balance = 0;
    }

    /**
     * Costruisce un conto bancario con un saldo assegnato.
     * @param anAccountNumber il numero di questo conto bancario
     * @param initialBalance il saldo iniziale
    */
    public BankAccount(int anAccountNumber, double initialBalance)
    {
        accountNumber = anAccountNumber;
        balance = initialBalance;
    }

    /**
     * Restituisce il numero di conto del conto bancario.
     * @return il numero di conto
    */
    public double getAccountNumber()
    {
        return accountNumber;
    }

    /**
     * Versa denaro nel conto bancario.
     * @param amount l'importo da versare
    */
    public void deposit(double amount)
    {
        double newBalance = balance + amount;
        balance = newBalance;
    }

    /**
     * Preleva denaro dal conto bancario.
     * @param amount l'importo da prelevare
    */
    public void withdraw(double amount)
    {
        double newBalance = balance - amount;
        balance = newBalance;
    }

    /**
     * Ispeziona il valore del saldo attuale del conto bancario.
     * @return il saldo attuale
    */
    public double getBalance()
    {
        return balance;
    }
}
```

## Esecuzione del programma

```
Size: 3
Expected: 3
First account number: 1008
Expected: 1008
Last account number: 1729
Expected: 1729
```

## Auto-valutazione

3. Come si costruisce un array di 10 stringhe? È un vettore di stringhe?
4. Cosa contiene `names` dopo l'esecuzione degli enunciati seguenti?

```
ArrayList<String> names = new ArrayList<String>();
names.add("A");
names.add(0, "B");
names.add("C");
names.remove(1);
```

## Errori comuni 7.3

### 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 proprio 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 7.2

### Miglioramenti per la sintassi di `ArrayList` in Java 7

La versione 7 di Java presenta alcune comode novità in merito alla sintassi utilizzabile con i vettori.

Quando si dichiara e 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”, perché le due parentesi angolari vuote hanno la forma di un diamante, `<>`.

Si possono anche fornire valori iniziali per gli elementi del vettore, in questo modo:

```
ArrayList<String> names = new ArrayList<>(["Ann", "Cindy", "Bob"]);
```

In Java 7, si può effettuare l'accesso agli elementi di un vettore usando l'operatore `[]` al posto dei metodi `get` e `set`. In sostanza, il compilatore traduce

```
String name = names[i];
names[i] = "Fred";
```

in

```
String name = names.get(i);
names.set(i, "Fred");
```

### 7.3 Classi involucro (*wrapper*) e auto-boxing

Per manipolare valori di un tipo primitivo come se fossero oggetti si usano le classi involucro.

Poiché in Java i numeri non sono oggetti, non potete inserirli direttamente all'interno di vettori: non potete, ad esempio, creare un oggetto di tipo `ArrayList<double>`. Per memorizzare sequenze di numeri in un vettore, dovete trasformarli in oggetti, usando le *classi involucro*. Esistono classi involucro per tutti gli otto tipi di dati primitivi:

| Tipo primitivo | Classe involucro |
|----------------|------------------|
| byte           | Byte             |
| boolean        | Boolean          |
| char           | Character        |
| double         | Double           |
| float          | Float            |
| int            | Integer          |
| long           | Long             |
| short          | Short            |

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`.

Ciascun oggetto di una classe involucro contiene un valore del corrispondente tipo primitivo. Ad esempio, un oggetto della classe `Double` contiene un valore di tipo `double`, come si può vedere nella Figura 7.

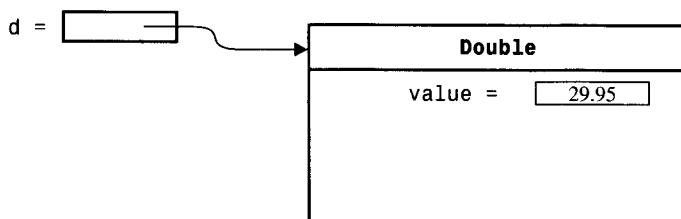
Gli oggetti involucro possono essere utilizzati al posto di valori di tipo primitivo ovunque sia richiesta la presenza di un oggetto. Ad esempio, si può memorizzare una sequenza di numeri in virgola mobile all'interno di un vettore di tipo `ArrayList<Double>`.

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 preferibile l'utilizzo, in inglese, di “auto-wrapping”).

Ad esempio, se si assegna un valore numerico a una variabile di tipo `Double`, questo viene automaticamente “inserito in una scatola”, cioè in un oggetto che funge da involucro.

**Figura 7**

Un oggetto di una classe involucro



```
Double d = 29.95; // auto-boxing: esattamente come se fosse
// Double d = new Double(29.95);
```

Viceversa, gli oggetti involucro vengono automaticamente “tolti dalla scatola” per generare valori di tipo primitivo, mediante un processo denominato *auto-unboxing*:

```
double x = d; // auto-unboxing: esattamente come double x = d.doubleValue();
```

Queste conversioni automatiche funzionano anche all’interno di espressioni aritmetiche. Ad esempio, questo codice

```
Double d = 29.95;
d = d + 1;
```

è perfettamente valido e il secondo enunciato significa:

- Converti **d** in un valore di tipo **double** mediante auto-unboxing
- Aggiungi 1
- Converti il risultato in un nuovo oggetto di tipo **Double** mediante auto-boxing
- Memorizza in **d** il riferimento all’oggetto involucro appena creato

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);
```

Tenete ben presente, però, che la memorizzazione di numeri in oggetti involucro è piuttosto inefficiente: l’utilizzo delle classi involucro è accettabile per sequenze di piccole dimensioni, ma per lunghe sequenze di numeri o di caratteri dovreste sempre utilizzare array.

## Auto-valutazione

5. Qual è la differenza tra i tipi **double** e **Double**?
6. Se **values** è un oggetto di tipo **ArrayList<Double>** con dimensione maggiore di zero, come si aggiunge un’unità al valore memorizzato nell’elemento di indice zero?



## 7.4 Il ciclo for esteso

La versione 5.0 del linguaggio Java ha introdotto un'abbreviazione sintattica molto utile per un tipo di ciclo frequentemente utilizzato. C'è spesso la necessità di scandire tutti gli elementi di una sequenza, ad esempio gli elementi di un array o di un vettore: il ciclo **for** esteso rende particolarmente semplice la programmazione di questa procedura.

Il ciclo **for** esteso scandisce tutti gli elementi di una raccolta.

Immaginate di voler sommare tutti gli elementi presenti nell'array `values`. Usando il ciclo **for** esteso potete fare in questo modo:

```
double[] values = ...;
double sum = 0;
for (double element : values)
{
    sum = sum + element;
}
```

Il corpo del ciclo viene eseguito per ciascun elemento presente nell'array `values`. All'inizio di ciascuna iterazione del ciclo, alla variabile `element` viene assegnato l'elemento successivo dell'array, quindi viene eseguito il corpo del ciclo: in pratica, potete immaginare che il ciclo si legga come “per ogni `element` appartenente a `values`”.

Vi potreste, a questo punto, chiedere perché Java non vi consenta di scrivere qualcosa di simile a “`for each (element in values)`”, che sarebbe senza dubbio più chiaro e simile a quanto avviene in altri linguaggi di programmazione. I progettisti del linguaggio Java hanno preso in seria considerazione questa opzione, ma questo costrutto sintattico è stato aggiunto a Java molti anni dopo la prima versione del linguaggio e l'aggiunta di due nuove parole riservate, `each` e `in`, avrebbe impedito la compilazione di tutti quei programmi, scritti precedentemente, che usano tali identificativi come nomi di variabile o di metodo (ad esempio, `System.in`).

Per scandire tutti gli elementi di un array non siete comunque costretti a utilizzare il ciclo **for** esteso; lo stesso ciclo può essere realizzato con un **for** normale e una variabile indice esplicita:

```
double[] values = ...;
double sum = 0;
for (int i = 0; i < values.length; i++)
{
    double element = values[i];
    sum = sum + element;
}
```

In un ciclo **for** esteso la variabile di ciclo contiene un elemento, non un indice.

Notate una differenza importante che esiste tra il ciclo **for** esteso e quello normale: nel primo caso, alla variabile di ciclo `element` viene assegnato il valore degli elementi, `values[0]`, `values[1]`, e così via, mentre nel secondo caso alla variabile di ciclo `i` vengono assegnati i valori `0, 1, ...`

Potete usare il ciclo **for** esteso anche per ispezionare tutti gli elementi di un vettore. Ad esempio, il ciclo seguente calcola il saldo totale di tutti i conti bancari:

```
ArrayList<BankAccount> accounts = ...;
```

## Sintassi di Java

### 7.3 Il ciclo for esteso

#### Sintassi

```
for (nomeTipo variabile : raccolta)
    enunciato
```

#### 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

```
for (double element : values)
{
    sum = sum + element;
```

Un array o un vettore

La variabile contiene  
un elemento, non un indice.

```
double sum = 0;
for (BankAccount account : accounts)
{
    sum = sum + account.getBalance();
}
```

Il ciclo è equivalente a questo ciclo normale:

```
ArrayList<BankAccount> accounts = ...;
double sum = 0;
for (int i = 0; i < accounts.size(); i++)
{
    BankAccount account = accounts.get(i);
    sum = sum + account.getBalance();
}
```

Il ciclo for esteso viene utilizzato per uno scopo ben preciso: la scansione, dall'inizio alla fine, di tutti gli elementi di una raccolta. Non è, quindi, adatto per tutti gli algoritmi che operano su array e, in particolare, non consente di modificare i contenuti dell'array. Questo ciclo, infatti, *non* riempie l'array di zeri:

```
for (double element : values)
{
    element = 0; // ERRORE: questa assegnazione
                  // non modifica gli elementi dell'array
}
```

Quando il ciclo viene eseguito, durante la 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'elemento `values[0]`. Il rimedio è semplice: usate un ciclo ordinario.

```

for (int i = 0; i < values.length; i++)
{
    values[i] = 0; // così va bene
}

```



## Auto-valutazione

7. Scrivete un ciclo `for` esteso che visualizzi tutti gli elementi dell'array `values`.
8. Cosa fa questo ciclo `for` esteso?

```

int counter = 0;
for (BankAccount a : accounts)
{
    if (a.getBalance() == 0) { counter++; }
}

```

## 7.5 Array riempiti solo in parte

Supponete di dover scrivere un programma che legge una sequenza di numeri e li scrive in un array. Quanti numeri inserirà l'utente? Non è molto ragionevole chiedere all'utente di contarli prima di introdurli, perché questo è proprio quel tipo di lavoro che l'utente si aspetta che venga eseguito da un computer. Sfortunatamente, siete di fronte a un problema: dovete impostare la dimensione dell'array prima di sapere quanti sono gli elementi di cui avete bisogno e, dopo aver impostato la dimensione dell'array, essa non può più essere modificata.

Per risolvere questo problema, a volte potete creare un array che sia sicuramente più grande del numero massimo possibile di voci, per poi riempirlo solo parzialmente. Per esempio, potete decidere che l'utente non inserirà mai più di 100 valori, per cui create un array di dimensione 100:

```

final int VALUES_LENGTH = 100;
double[] values = new double[VALUES_LENGTH];

```

Insieme a un array riempito parzialmente, usate una variabile ausiliaria che tenga traccia del numero di elementi realmente utilizzati.

dopodiché usate una variabile ausiliaria che dica quanti elementi dell'array sono realmente utilizzati. Solitamente si attribuisce a tale variabile ausiliaria un nome ottenuto aggiungendo il suffisso `Size` al nome dell'array:

```
int valuesSize = 0;
```

Ora, `values.length` è la capacità dell'array `values`, mentre `valuesSize` è la dimensione reale dell'array, come si può vedere in Figura 8. Continuando ad aggiungere elementi all'array, bisogna incrementare di pari passo la variabile `valuesSize`:

```

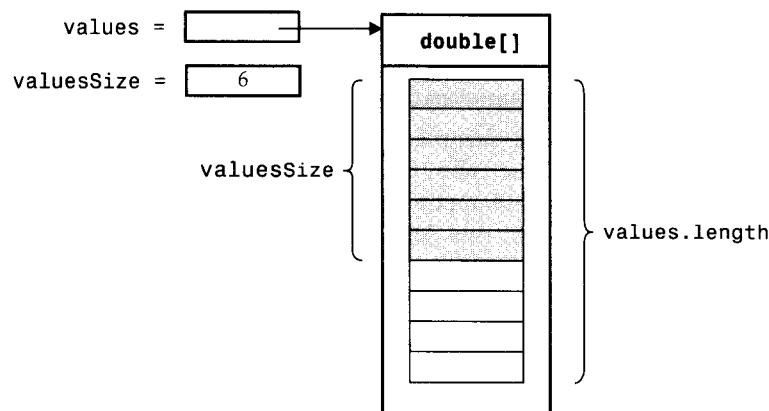
values[valuesSize] = x;
valuesSize++;

```

In questo modo, `valuesSize` contiene sempre il conteggio esatto degli elementi.

Il frammento di codice seguente mostra come leggere numeri, inserendoli in un array riempito solo in parte.

**Figura 8**  
Un array riempito solo in parte



```
int valuesSize = 0;
Scanner in = new Scanner(System.in);
while (in.hasNextDouble())
{
    if (valuesSize < values.length)
    {
        values[valuesSize] = in.nextDouble();
        valuesSize++;
    }
}
```

Al termine di questo ciclo, `valuesSize` contiene il numero di elementi effettivamente presenti nell'array. Notate che, nel momento in cui la variabile ausiliaria `valuesSize` raggiunge il valore pari alla lunghezza dell'array, occorre interrompere la lettura di dati: il Paragrafo 7.6 propone una soluzione per superare questa limitazione, aumentando, di fatto, la dimensione dell'array.

Per elaborare gli elementi così acquisiti, usate nuovamente la variabile ausiliaria al posto della lunghezza dell'array. Questo ciclo visualizza il contenuto dell'array riempito solo in parte:

```
for (int i = 0; i < valuesSize; i++)
{
    System.out.println(values[i]);
}
```

I vettori, esemplari di `ArrayList`, usano proprio questa tecnica: un vettore contiene un array di oggetti e, quando l'array non ha più spazio disponibile, il vettore stesso crea un array più grande e vi copia i propri elementi. Tutto ciò, però, accade all'interno dei metodi del vettore, per cui non ve ne dovete mai occupare direttamente.



### Auto-valutazione

9. Scrivete un ciclo che visualizzi in ordine inverso, partendo dall'ultimo, gli elementi dell'array `values` riempito solo in parte.
10. Come si elimina l'ultimo elemento presente nell'array `values` riempito solo in parte?
11. Per quale motivo i programmati usano un array riempito solo in parte per memorizzare numeri, invece di un vettore?



## Errori comuni 7.4

### 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. La causa di problemi più frequente è l'uso di array di dimensioni fisse. Supponete di scrivere un programma che cerchi una stringa in un file. Inserite ciascuna riga del file in una stringa e create un array di stringhe. Di quali dimensioni lo create? Sicuramente nessuno userà il vostro programma con un testo di 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 su Internet). All'improvviso il vostro programma si trova ad avere a che fare con decine o centinaia di migliaia di righe. Che cosa farà? Ce la farà a gestire tutti quei dati? Respingerà educatamente tutto quello che è di troppo? Crollerà incenerito?

Un famoso articolo (Barton P. Miller, Louis Fericksen e Bryan So, "An Empirical Study of the Reliability of Unix Utilities", *Communications of the ACM*, vol. 33, n. 12, pagg. 32–44) analizzava la reazione di diversi programmi del sistema operativo UNIX di fronte a quantità di dati enormi o casuali. Tristemente, circa un quarto dei programmi collaudati non si comportava affatto bene, terminando bruscamente o bloccandosi senza nemmeno un messaggio di errore ragionevole. Per esempio, in alcune versioni di UNIX, il programma di backup su nastro, *tar*, non riesce a gestire nomi di file più lunghi di 100 caratteri, che è un limite piuttosto irragionevole. Molti di questi inconvenienti sono causati da caratteristiche del linguaggio C, che, a differenza di Java, rende difficile la memorizzazione di stringhe di dimensione arbitraria.

## 7.6 Semplici algoritmi per vettori e array

In questo paragrafo vedrete alcuni degli algoritmi più utilizzati per l'elaborazione di array e vettori.

Gli esempi saranno riferiti indifferentemente ad array e a vettori, in modo che prendiate confidenza con la sintassi relativa a entrambe queste strutture.

### 7.6.1 Riempimento

Questo ciclo riempie di zeri un array:

```
for (int i = 0; i < values.length; i++)
{
    values[i] = 0;
}
```

Qui, invece, riempiamo un vettore con i quadrati dei primi numeri interi. Notate che l'elemento di indice 0 contiene  $0^2$ , l'elemento di indice 1 contiene  $1^2$  e così via.

```
for (int i = 0; i < values.size(); i++)
{
    values.set(i, i * i);
}
```

## 7.6.2 Calcolare somme e valori medi

Per calcolare la somma di tutti gli elementi, si usa semplicemente un totale parziale.

```
double total = 0;
for (double element : values)
{
    total = total + element;
}
```

Per ottenere il valore medio, dividiamo per il numero di elementi:

```
double average = total / values.size(); // per un vettore
```

Accertatevi, ovviamente, che la dimensione non sia zero.

## 7.6.3 Contare valori aventi determinate caratteristiche

Supponete di voler sapere quanti conti correnti con determinate caratteristiche sono presenti in una banca. Per risolvere il problema dovete scandire l'intero vettore e incrementare un contatore ogni volta che trovate un elemento che soddisfa la condizione. In questo esempio contiamo il numero di conti correnti il cui saldo è maggiore o uguale a un valore di soglia:

```
public class Bank
{
    private ArrayList<BankAccount> accounts;

    public int count(double atLeast)
    {
        int matches = 0;
        for (BankAccount account : accounts)
        {
            if (account.getBalance() >= atLeast)
                matches++; // trovato
        }
        return matches;
    }
    ...
}
```

## 7.6.4 Trovare il valore massimo o minimo

Immaginate di voler trovare il conto avente il saldo maggiore tra quelli presenti nella banca. Scegliete un candidato come valore massimo: se trovate un elemento con valore maggiore, sostituite il candidato con tale valore. Raggiunta la fine della sequenza, avete trovato il massimo.

C'è soltanto un problema: quando ispezionate il primo elemento, non avete ancora un candidato per il valore massimo. Il modo più semplice per risolvere il problema è usare come primo candidato proprio l'elemento iniziale, iniziando i confronti dall'elemento successivo.

**Per contare i valori presenti in un vettore e aventi determinate caratteristiche, ispezionate tutti gli elementi e contate quelli che rispondono ai requisiti.**

**Per trovare il valore massimo (o minimo), inizializzate un candidato con l'elemento iniziale, quindi confrontatelo con gli elementi rimanenti e aggiornatelo se trovate un elemento maggiore (o minore).**

```

BankAccount largestYet = accounts.get(0);
for (int i = 1; i < accounts.size(); i++)
{
    BankAccount a = accounts.get(i);
    if (a.getBalance() > largestYet.getBalance())
        largestYet = a;
}
return largestYet;

```

In questo caso usiamo un ciclo `for` ordinario, perché non ispezioniamo tutti gli elementi, come facevamo nei casi precedenti: il primo elemento non viene analizzato.

Ovviamente, questo metodo funziona soltanto se il vettore contiene almeno un elemento: non ha molto senso cercare l'elemento di valore maggiore in un insieme vuoto. In tal caso possiamo restituire `null`:

```

if (accounts.size() == 0) return null;
BankAccount largestYet = accounts.get(0);
...

```

Negli Esercizi R7.5 e R7.6 rivedrete questo algoritmo, un po' modificato.

Per calcolare il valore minimo di un insieme di dati, scegliete un candidato per il minimo e aggiornatelo ogni volta che trovate un valore *inferiore*. Terminata la sequenza, avete trovato il minimo.

## 7.6.5 Trovare un valore

Per trovare un valore, controllate tutti gli elementi finché non trovate quello cercato.

Supponete di voler sapere se nella vostra banca è presente un conto con un determinato numero identificativo: semplicemente, esaminete ciascun elemento fino a trovare quello cercato oppure fino ad arrivare alla fine della sequenza. Notate che, se nessuno dei conti corrisponde a quello cercato, il ciclo di ricerca potrebbe avere esito negativo. Questo procedimento si chiama *ricerca lineare*.

```

public class Bank
{
    public BankAccount find(int accountNumber)
    {
        for (BankAccount account : accounts)
        {
            if (account.getAccountNumber() == accountNumber)
                return account; // trovato
        }
        return null; // non trovato nell'intero vettore
    }
    ...
}

```

Notate che, se non trova il conto cercato, il metodo restituisce `null`.

## 7.6.6 Trovare la posizione di un elemento

Spesso occorre trovare la posizione di un elemento all'interno di una sequenza, per poterlo sostituire o eliminare: si usa una variante dell'algoritmo di ricerca lineare, tenendo

traccia della posizione anziché dell'elemento che soddisfa la ricerca. Ecco come trovare la posizione del primo elemento di valore superiore a 100.

```

int pos = 0;
boolean found = false;
while (pos < values.size() && !found)
{
    if (values.get(pos) > 100)
    {
        found = true;
    }
    else
    {
        pos++;
    }
}
if (found) { System.out.println("Position: " + pos); }
else { System.out.println("Not found"); }

```

### 7.6.7 Eliminare un elemento

La rimozione di un elemento da un vettore è estremamente semplice: si usa il metodo `remove`. Con un array, invece, dobbiamo lavorare più duramente.

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 7.5.

Se gli elementi si trovano nell'array senza alcuna particolare proprietà di ordinamento posizionale, basta semplicemente sovrascrivere l'elemento da eliminare con l'*ultimo* elemento presente nell'array, decrementando la variabile che memorizza la dimensione dell'array: la procedura è illustrata nella Figura 9.

```

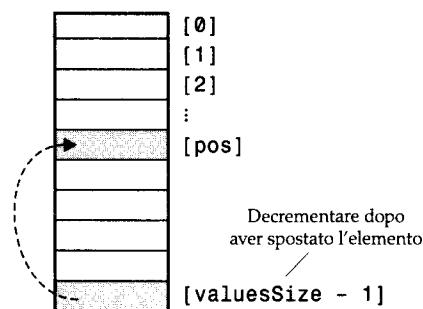
values[pos] = values[valuesSize - 1];
valuesSize--;

```

Se, invece, l'ordine tra gli elementi è importante, 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 10.

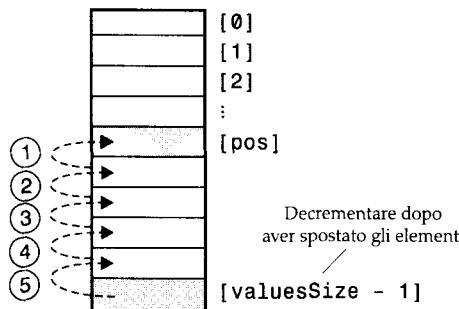
**Figura 9**

Rimozione di un elemento  
da un array non ordinato



**Figura 10**

Rimozione di un elemento da un array ordinato



```
for (int i = pos; i < valuesSize - 1; i++)
{
    values[i] = values[i + 1];
}
valuesSize--;
```

### 7.6.8 Inserire un elemento

Per inserire un elemento in un vettore si usa il metodo `add`. Vedremo ora, invece, 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 7.5.

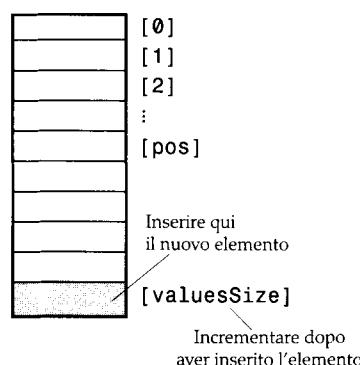
Se l'ordine tra gli elementi non è importante, potete semplicemente inserire i nuovi elementi alla fine, incrementando la variabile che memorizza la dimensione dell'array (Figura 11).

```
if (valuesSize < values.length)
{
    values[valuesSize] = newElement;
    valuesSize++;
}
```

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 vi interessa vanno spostati di un posto, in una posizione avente indice maggiore di uno rispetto all'attuale, per poi, al termine della procedura, inserire il nuovo elemento nella posizione rimasta “libera”, come si vede nella Figura 12.

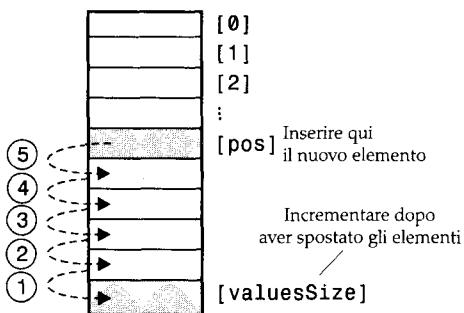
**Figura 11**

Inserimento di un elemento in un array non ordinato



**Figura 12**

Inserimento di un elemento  
in un array ordinato



Osservate con attenzione l'ordine in cui vengono effettuati gli spostamenti. Quando si rimuove 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 fin quando si raggiunge la posizione desiderata.

```
if (valuesSize < values.length)
{
    for (int i = valuesSize; i > pos; i--)
    {
        values[i] = values[i - 1];
    }
    values[pos] = newElement;
    valuesSize++;
}
```

### 7.6.9 Copiare e ingrandire array

Una variabile di tipo array  
memorizza un riferimento all'array.  
Copiando la variabile si ottiene  
un secondo riferimento  
al medesimo array.

```
double[] values = new double[6];
... // riempimento dell'array
double[] prices = values;
```

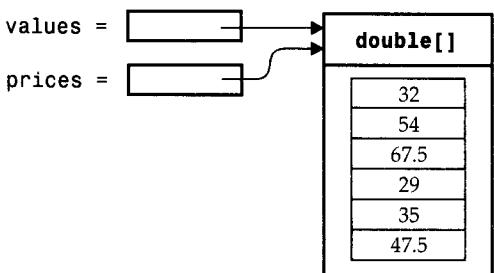
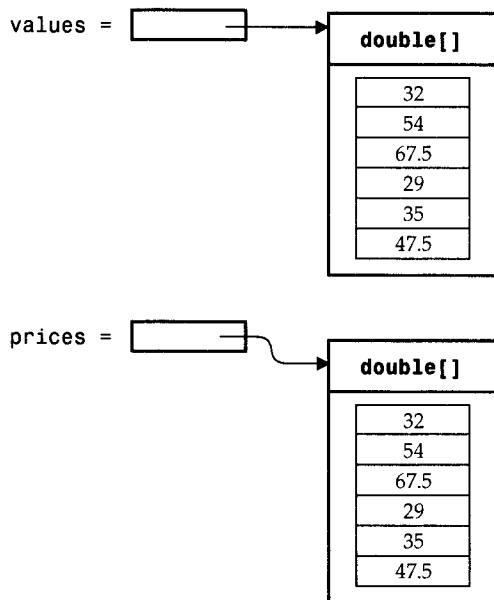
Per copiare gli elementi  
di un array, usate il metodo  
Arrays.copyOf.

```
double[] prices = Arrays.copyOf(values, values.length);
```

`Arrays.copyOf` si può anche usare per ingrandire un array che si sia riempito. Questo enunciato ha l'effetto di raddoppiare la dimensione di un array, come si vede nella Figura 14:

```
values = Arrays.copyOf(values, 2 * values.length);
```

Ecco, ad esempio, come memorizzare in un array una lunga sequenza di numeri di lunghezza non nota a priori:

(1) Dopo l'assegnamento `prices = values`(2) Dopo la chiamata `Arrays.copyOf`**Figura 13**

Differenza tra copiatura di un riferimento ad array e copiatura di un array

```
int valuesSize = 0;
while (in.hasNextDouble())
{
    if (valuesSize == values.length)
        values = Arrays.copyOf(values, 2 * values.length);
    values[valuesSize] = in.nextDouble();
    valuesSize++;
}
```

### 7.6.10 Visualizzare gli elementi con separatori

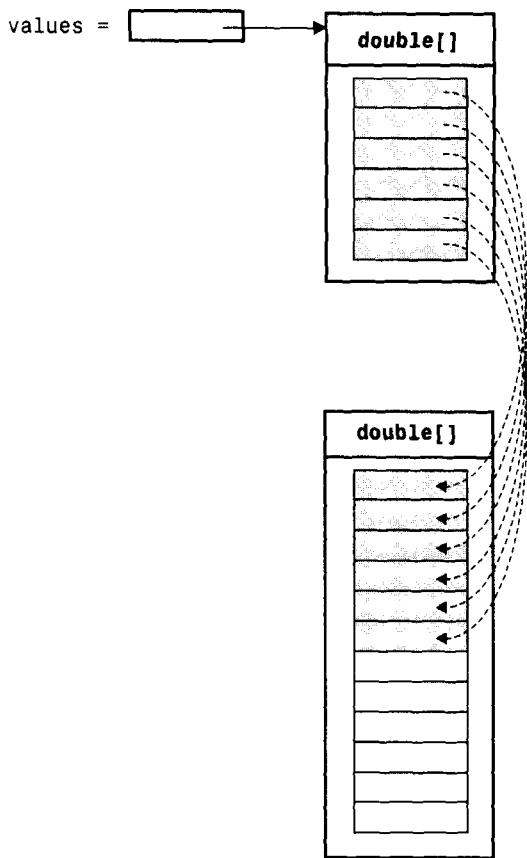
Quando si visualizzano gli elementi di un array o di un vettore, solitamente li si vuole separare, spesso con virgole o barrette verticali, in questo modo:

Ann | Bob | Cindy

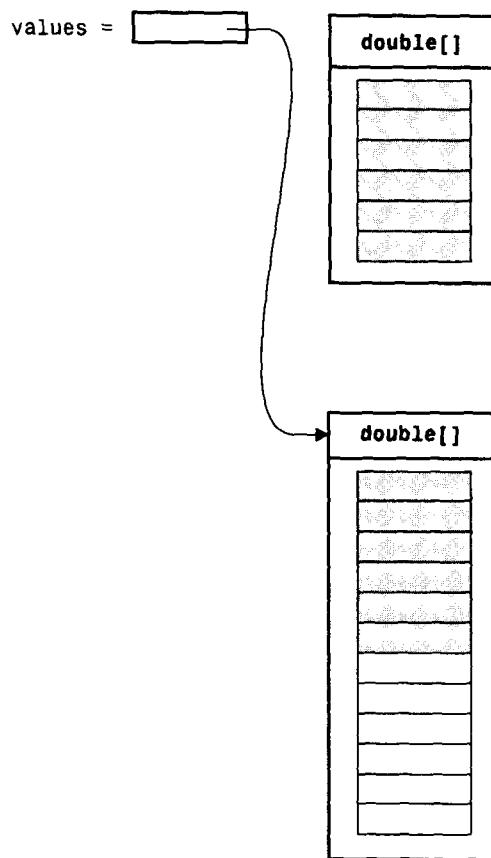
Osservate che il numero di separatori è inferiore di un'unità rispetto al numero di elementi. Si visualizza un separatore prima di ogni elemento, *tranne il primo* (che è quello di indice 0):

```
for (int i = 0; i < names.size(); i++)
{
    if (i > 0)
    {
        System.out.print(" | ");
    }
    System.out.print(names.get(i));
}
```

- ① Copiare gli elementi in un array più grande



- ② Memorizzare il riferimento all'array più grande in `values`



**Figura 14** L'esempio che segue è un programma che realizza una classe `Bank` utilizzando un vettore di conti bancari. I metodi della classe `Bank` usano alcuni degli algoritmi che avete visto in questo paragrafo.

#### File ch07/bank/Bank.java

```
import java.util.ArrayList;

/**
 * Questa banca contiene conti bancari.
 */
public class Bank
{
    private ArrayList<BankAccount> accounts;

    /**
     * Costruisce una banca priva di conti bancari.
     */
    public Bank()
```

```
{  
    accounts = new ArrayList<BankAccount>();  
}  
  
/**  
 * Aggiunge un conto bancario a questa banca.  
 * @param a il conto da aggiungere  
 */  
public void addAccount(BankAccount a)  
{  
    accounts.add(a);  
}  
  
/**  
 * Restituisce la somma dei saldi di tutti i conti della banca.  
 * @return la somma dei saldi  
 */  
public double getTotalBalance()  
{  
    double total = 0;  
    for (BankAccount a : accounts)  
    {  
        total = total + a.getBalance();  
    }  
    return total;  
}  
  
/**  
 * Conta il numero di conti bancari aventi saldo maggiore  
 * o uguale al valore indicato.  
 * @param atLeast il saldo minimo perché un conto venga conteggiato  
 * @return il numero di conti aventi saldo maggiore o uguale  
 *         al saldo indicato  
 */  
public int countBalancesAtLeast(double atLeast)  
{  
    int matches = 0;  
    for (BankAccount a : accounts)  
    {  
        if (a.getBalance() >= atLeast) matches++; // trovato  
    }  
    return matches;  
}  
  
/**  
 * Verifica se la banca contiene un conto avente il numero indicato.  
 * @param accountNumber il numero di conto da cercare  
 * @return il conto con il numero indicato,  
 *         oppure null se tale conto non esiste  
 */  
public BankAccount find(int accountNumber)  
{  
    for (BankAccount a : accounts)  
    {  
        if (a.getAccountNumber() == accountNumber) // trovato  
            return a;  
    }  
}
```

```

        return null; // non trovato nell'intero vettore
    }

    /**
     * Restituisce il conto bancario avente il saldo maggiore.
     * @return il conto con il saldo maggiore, oppure null se
     *         la banca non ha conti
    */
    public BankAccount getMaximum()
    {
        if (accounts.size() == 0) return null;
        BankAccount largestYet = accounts.get(0);
        for (int i = 1; i < accounts.size(); i++)
        {
            BankAccount a = accounts.get(i);
            if (a.getBalance() > largestYet.getBalance())
                largestYet = a;
        }
        return largestYet;
    }
}

```

### File ch07/bank/BankTester.java

```

/**
 * Questo programma collauda la classe Bank.
 */
public class BankTester
{
    public static void main(String[] args)
    {
        Bank firstBankOfJava = new Bank();
        firstBankOfJava.addAccount(new BankAccount(1001, 20000));
        firstBankOfJava.addAccount(new BankAccount(1015, 10000));
        firstBankOfJava.addAccount(new BankAccount(1729, 15000));

        double threshold = 15000;
        int c = firstBankOfJava.countBalancesAtLeast(threshold);
        System.out.println("Count: " + c);
        System.out.println("Expected: 2");

        int accountNumber = 1015;
        BankAccount account = firstBankOfJava.find(accountNumber);
        if (account == null)
            System.out.println("No matching account");
        else
            System.out.println("Balance of matching account: "
                + account.getBalance());
        System.out.println("Expected: 10000");

        BankAccount max = firstBankOfJava.getMaximum();
        System.out.println("Account with largest balance: "
            + max.getAccountNumber());
        System.out.println("Expected: 1001");
    }
}

```

## Esecuzione del programma

```
Count: 2
Expected: 2
Balance of matching account: 10000.0
Expected: 10000
Account with largest balance: 1001
Expected: 1001
```



## Auto-valutazione

12. Come si comporta il metodo `find` se esistono due conti bancari con numero di conto uguale al numero cercato?
13. Si potrebbe usare un ciclo `for` esteso nel metodo `getMaximum`?
14. Nella visualizzazione con separatori, abbiamo evitato di scrivere il separatore prima dell'elemento iniziale. Riscrivete il ciclo in modo che il separatore venga visualizzato *dopo* ogni elemento, con eccezione dell'ultimo.
15. Qual è il problema relativo a questa soluzione, proposta come alternativa all'algoritmo visto nel Paragrafo 7.6.10?  

```
System.out.print(names.get(0));
for (int i = 1; i < names.size(); i++)
    System.out.print(" | " + names.get(i));
```



## Consigli per la produttività 7.1

### Visualizzare array e vettori con facilità

Se `values` è un array, l'espressione

```
Arrays.toString(values)
```

restituisce una stringa che ne descrive gli elementi, usando questo formato:

```
[32, 54, 67.5, 29, 35, 47.5]
```

Gli elementi sono racchiusi tra una coppia di parentesi quadre e separati da virgole. Si tratta di una funzionalità che può essere comoda in fase di debugging:

```
System.out.println("values=" + Arrays.toString(values));
```

Con un vettore la cosa è ancora più semplice, basta passare il vettore stesso al metodo `println`:

```
System.out.println(names); // Visualizza [Ann, Bob, Cindy]
```



## Consigli pratici 7.1

### Usare array e vettori

Per elaborare una sequenza di valori, solitamente è necessario un array o un vettore (in situazioni molto semplici, però, è possibile elaborare i dati mentre vengono ac-

quisiti, senza doverli memorizzare): qui vediamo il procedimento da seguire, passo dopo passo.

Consideriamo, come esempio, questo problema: avendo a disposizione i risultati di alcune prove d'esame di uno studente, dobbiamo calcolare il risultato finale, che è la somma di tutti i risultati parziali, con l'esclusione del voto più basso. Ad esempio, se i voti sono

8 7 8.5 9.5 7 5 10

il voto finale è 50.

Se, però, c'è un solo voto, sarebbe davvero crudele eliminarlo: in tal caso, quell'unico voto sarà anche il voto finale. Infine, se non è disponibile alcun voto, il voto finale dovrebbe essere zero.

#### Fase 1 Scomponete il problema in sottoproblemi

Di solito il problema da risolvere viene scomposto in più sottoproblemi, ad esempio così

- Leggere i dati, scrivendoli in un array o in un vettore.
- 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 7.6: 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, elimineremo 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:

Leggere i dati in ingresso.

Eliminare il valore minimo.

Calcolare la somma.

#### Fase 2 Scegliete tra array e vettori

In generale, i vettori sono più comodi degli array. Dovreste, però, scegliere un array se è verificata una di queste condizioni.

- Conoscete a priori il numero di elementi da memorizzare e, in seguito, la dimensione della sequenza non cambia.
- Dovete memorizzare una lunga sequenza di numeri.

Nessuna di queste condizioni si applica al nostro esempio, per cui memorizzeremo i voti in un vettore (nella cartella ch07/scores2 del pacchetto di file scaricabili per questo libro troverete una soluzione alternativa, che usa array).

**Fase 3** Individuate gli algoritmi che vi servono

A volte capita che uno dei passi individuati nella Fase 1 corrisponda esattamente a uno degli algoritmi elementari, come avviene, nel nostro caso, con il calcolo della somma finale. Altre volte occorrerà combinare più algoritmi: nel nostro esempio, per eliminare il voto minimo, dobbiamo trovare il valore minimo (Paragrafo 7.6.4), individuare la sua posizione (Paragrafo 7.6.6) e, infine, eliminare l'elemento che si trova in quella posizione (Paragrafo 7.6.7).

Aumentiamo, quindi, il livello di dettaglio dello pseudocodice, in questo modo:

- Leggere i dati in ingresso.
- Trovare il valore minimo.
- Trovare la sua posizione.
- Eliminare il valore minimo.
- Calcolare la somma.

Questo progetto funziona, ma lo si può migliorare: è più semplice calcolare la somma di tutti i valori e, poi, sottrarre il minimo al totale ottenuto. In questo modo possiamo evitare di determinare la posizione del minimo e di eliminarlo:

- Leggere i dati in ingresso.
- Trovare il valore minimo.
- Calcolare la somma.
- Sottrarre il valore minimo.

**Fase 4** Strutturate il programma mediante 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, come `DataSet` nel Capitolo 6 o `Bank` nel paragrafo precedente.

Nel nostro esempio, memorizziamo i voti all'interno della classe `GradeBook` (“registro scolastico”).

```
public class GradeBook
{
    private ArrayList<Double> scores;
    ...
    public void addScore(double score) { ... }
    public double finalScore() { ... }
}
```

Un'altra classe, `ScoreAnalyzer`, avrà il compito di leggere i dati introdotti dall'utente e di visualizzare il risultato. Il suo metodo `main` invoca i metodi di `GradeBook`:

```
GradeBook book = new GradeBook();
System.out.println("Please enter values, Q to quit:");
while (in.hasNextDouble())
{
```

```

        book.addScore(in.nextDouble());
    }
System.out.println("Final score: " + book.finalScore());

```

A questo punto, al metodo `finalScore` rimane gran parte del lavoro, ma non è giusto che tutto sia compito suo: definiamo, invece, i metodi ausiliari

```

public double sum()
public double minimum()

```

che realizzano, rispettivamente, gli algoritmi visti nel Paragrafo 7.6.2 e 7.6.4. Infine, possiamo scrivere il metodo `finalScore`:

```

public double finalScore()
{
    if (scores.size() == 0)
        return 0;
    else if (scores.size() == 1)
        return scores.get(0);
    else
        return sum() - minimum();
}

```

#### Fase 5 Completate e collaudate il programma

Completate la scrittura delle vostre classi e collaudatele, come visto nei Consigli pratici 3.1. Riguardate il codice che avete scritto e verificate di aver gestito sia i casi normali sia le condizioni eccezionali. Cosa accade con un array o un vettore 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 restituire il voto speciale zero *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 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)  | 0                  | Un registro di voti vuoto genera il voto finale zero. |

Il programma completo si trova nella cartella `ch07/scores` del pacchetto di file scaricabili per questo libro.



## Esempi completi 7.1

### Tirare i dadi

Dovete analizzare l'equità di un dado contando la frequenza di apparizione dei suoi valori: 1, 2, ..., 6. La sequenza di ingresso del programma sarà costituita dalle osservazioni di lanci del dato e dovrete produrre in uscita una tabella contenente la frequenza di ciascun valore osservato.

#### Fase 1 Scomponete il problema in sottoproblemi

Il primo tentativo di scomposizione deriva semplicemente dalla lettura dell'enunciazione del problema da risolvere:

- Leggere i valori ottenuti con i lanci del dado.
- Contare il numero di occorrenze dei valori 1, 2, ..., 6.
- Visualizzare i conteggi.

Ragioniamo però ancora un po' sul problema. Questa scomposizione suggerisce di leggere e memorizzare tutti i valori ottenuti con i lanci del dado, ma abbiamo veramente bisogno di memorizzarli? Dopo tutto, vogliamo soltanto sapere quanto spesso compare ciascuna faccia del dado: se usiamo un array o un vettore di contatori, possiamo dimenticarci di ciascun valore letto subito dopo aver incrementato il relativo contatore.

Ottieniamo, quindi, questo miglioramento:

- Per ogni valore di ingresso
- Incrementare il contatore corrispondente.
- Visualizzare il contenuto dei contatori.

#### Fase 2 Scegliete tra array e vettori

Non c'è alcun bisogno di memorizzare i valori letti in ingresso. Dobbiamo soltanto decidere come memorizzare i contatori, il cui numero è noto a priori: ci serve un contatore per ognuna delle sei facce del dado. Dato che il numero di elementi della sequenza da memorizzare è noto a priori, usiamo un array.

Usiamo, però, un array `counters` contenente sette numeri interi, “sprecando” così l'elemento `counters[0]`. Questo “trucco” agevola l'aggiornamento dei contatori, perché, dopo aver letto un valore, `value`, compreso tra 1 e 6, eseguiamo semplicemente

```
counters[value]++; // value è compreso tra 1 e 6
```

Di conseguenza, dichiariamo l'array in questo modo:

```
int[] counters = new int[sides + 1];
```

Perché abbiamo introdotto la variabile `sides`, che ha come valore il numero di facce del dado? In questo modo, se in futuro vorrete fare esperimenti con un dado a dodici facce, sarà sufficiente modificare il valore di questa variabile.

### Fase 3 Individuate gli algoritmi che vi servono

Abbiamo già discusso ampiamente l'algoritmo di acquisizione dei valori in ingresso: si leggono i valori, uno dopo l'altro, e si incrementa il contatore corrispondente al valore letto.

Dobbiamo soltanto decidere come visualizzare i conteggi finali, ad esempio in questo modo:

```
1: 3  
2: 3  
3: 2  
4: 2  
5: 2  
6: 0
```

Non abbiamo mai visto un algoritmo che produca esattamente questo formato, anche se è simile al ciclo elementare che visualizza tutti gli elementi di un array:

```
for (int element : counters)  
{  
    System.out.println(element);  
}
```

Questo ciclo, però, non è adeguato, per due motivi. Innanzitutto, visualizza l'elemento di indice zero, che in questo programma non è utilizzato: quindi, se vogliamo evitare di visualizzare tale valore, il ciclo `for` esteso non è adatto. Ci serve un ciclo `for` tradizionale:

```
for (int i = 1; i < counters.length; i++)  
{  
    System.out.printf("%2d: %4d\n", i, counters[i]);  
}
```

### Fase 4 Strutturate il programma mediante classi e metodi

Come in Consigli pratici 6.1, useremo una classe `DiceCounter` che memorizza ed elabora i dati e un'altra classe, `DiceAnalyzer`, che legge i dati forniti dall'utente. La classe `DiceCounter` gestisce i contatori e ha due metodi:

```
public void addValue(int value)  
  
public void display()
```

Il metodo `addValue` incrementa un contatore, mentre il metodo `display` visualizza tutti i conteggi, come visto in precedenza.

Il metodo `main` della classe legge i valori in ingresso e invoca `addValue` per ciascun valore letto, quindi invoca `display`.

**Fase 5** Completate e collaudate il programma

Al termine di questa sezione trovate il codice completo del programma, dove compare una caratteristica importante di cui non abbiamo mai parlato. Quando aggiorniamo un contatore, in questo modo

```
counters[value]++;
```

vogliamo essere sicuri che l'utente non abbia fornito un valore errato, che provocherebbe un'eccezione per errore di limiti nell'array. Dobbiamo, quindi, rifiutare valori minori di uno o maggiori di `sides`.

La tabella che segue elenca casi di prova e i risultati previsti. Per risparmiare spazio, nella colonna "risultato previsto" mostriamo 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 ch07/dice/DiceCounter.java**

```
/**
 * Questa classe conta le occorrenze delle facce di un dado.
 */
public class DiceCounter
{
    private int[] counters;

    /**
     * Costruisce una struttura di conteggio adatta a un
     * dado con il numero di facce richiesto.
     * @param sides il numero di facce del dado oggetto degli esperimenti
     */
    public DiceCounter(int sides)
    {
        counters = new int[sides + 1];
    }

    /**
     * Aggiunge un valore.
     * @param value il valore, che deve essere compreso tra 1 e il numero di facce
     */
    public void addValue(int value)
    {
```

```
        counters[value]++;
    }

    /**
     * Visualizza tutti i conteggi registrati.
    */
    public void display()
    {
        for (int j = 1; j < counters.length; j++)
        {
            System.out.println(j + ": " + counters[j])
        }
    }
}
```

### File ch07/dice/DiceAnalyzer.java

```
import java.util.Scanner;

public class DiceAnalyzer
{
    public static void main(String[] args)
    {
        final int SIDES = 6;
        DiceCounter counter = new DiceCounter(SIDES);
        System.out.println("Please enter values, Q to quit:");
        Scanner in = new Scanner(System.in);
        while (in.hasNextInt())
        {
            int value = in.nextInt();
            if (1 <= value && value <= SIDES)
            {
                counter.addValue(value);
            }
            else
            {
                System.out.println(value + " is not a valid input.");
            }
        }
        counter.display();
    }
}
```

### 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
```

## 7.7 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.**

Di fatto, creare un nuovo caso di prova quando 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, ma usatelo per collaudare la versione successiva del programma e tutte le versioni seguenti. Una simile raccolta di casi di prova è detta *pacchetto di prove* (“*test suite*”).

Vi stupirete della frequenza con cui un errore, già corretto, riappaie 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 “tampone”. 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 di una serie di errori rilevati in passato si chiama collaudo regressivo (*regression testing*).

Come si organizza un pacchetto di prove? Il modo più semplice consiste nella scrittura di diverse classi di collaudo, come `BankTester1`, `BankTester2`, e così via.

Un altro approccio efficace prevede di realizzare una classe di collaudo generale e di fornirle dati in ingresso prelevati da file diversi. Consideriamo, ad esempio, questa classe di collaudo per la classe `Bank` del Paragrafo 7.6:

### File ch07/regression/BankTester.java

```
import java.util.Scanner;

/**
 * Questo programma collauda la classe Bank.
 */
public class BankTester
{
    public static void main(String[] args)
    {
        Bank firstBankOfJava = new Bank();
        firstBankOfJava.addAccount(new BankAccount(1001, 20000));
        firstBankOfJava.addAccount(new BankAccount(1015, 10000));
        firstBankOfJava.addAccount(new BankAccount(1729, 15000));

        Scanner in = new Scanner(System.in);

        double threshold = in.nextDouble();
        int c = firstBankOfJava.countBalancesAtLeast(threshold);
        System.out.println("Count: " + c);
        int expectedCount = in.nextInt();
        System.out.println("Expected: " + expectedCount);

        int accountNumber = in.nextInt();
        BankAccount a = firstBankOfJava.find(accountNumber);
```

```

        if (a == null)
            System.out.println("No matching account");
        else
        {
            System.out.println("Balance of matching account: "
                + a.getBalance());
            int matchingBalance = in.nextInt();
            System.out.println("Expected: " + matchingBalance);
        }
    }
}

```

Invece di usare valori prefissati per la soglia e per il numero di conto che si vuole cercare, il programma legge tali valori e le relative risposte previste. Eseguendo il programma con diversi valori di ingresso, si possono collaudare situazioni diverse, come, ad esempio, quelle necessarie per evidenziare gli eventuali errori per scarto di uno citati negli Errori comuni 6.2.

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 ch07/regression/input1.txt

```

15000
2
1015
10000

```

L'interfaccia per la riga di comando della maggior parte dei sistemi operativi fornisce gli strumenti per collegare un file al flusso di ingresso di un programma, come se i caratteri del file venissero effettivamente digitati sulla tastiera da un utente. Scrivete questo comando in una finestra di console:

```
java BankTester < 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 in ingresso dal file `input1.txt`. Questo processo viene chiamato “redirezione del flusso di ingresso”.

Nella finestra viene quindi visualizzato quanto segue:

### Esecuzione del programma

```

Count: 2
Expected: 2
Balance of matching account: 10000.0
Expected: 10000

```

È possibile redirigere anche il flusso di uscita. Per memorizzare in un file ciò che viene visualizzato sul flusso di uscita di un programma, usate questo comando:

```
java BankTester < input1.txt > output1.txt
```

Questa modo di operare è utile per archiviare i risultati dell'esecuzione di un pacchetto di prove.



## Auto-valutazione

16. 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?
17. Ipotizzate che un acquirente di un vostro programma scopra un errore: cosa dovreste fare prima di eliminare l'errore?
18. Perché il programma `BankTester` non chiede dati in ingresso?



## Consigli per la produttività 7.2

### 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 una serie di comandi automaticamente. Per esempio, immaginate di dover collaudare un programma mediante l'esecuzione di tre classi di collaudo:

```
java BankTester1
java BankTester2
java BankTester3 < input1.txt
```

In seguito, vi succede di scoprire un errore, lo correggete ed eseguite nuovamente le prove. A questo punto, avete bisogno di digitare ancora una volta i tre comandi: ci aspettiamo che debba esistere un sistema migliore per farlo. In ambiente Windows, inserite i comandi in un file di testo, che chiamerete `test.bat`:

### File test.bat

```
java BankTester1
java BankTester2
java BankTester3 < input1.txt
```

Quindi, per eseguire automaticamente i tre comandi contenuti in tale *file batch*, è sufficiente digitare il comando seguente:

`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 di 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.



## Note di cronaca 7.2

### Le vicende del Therac-25

Il Therac-25 è un dispositivo computerizzato per la terapia a emissione di radiazioni, destinata ai malati di tumore. Fra il giugno 1985 e il 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, pagg. 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 possibile rintracciarlo. Nessuno, fra i dipendenti della società che furono interrogati, fu in grado di dire alcunché circa il livello di studi o le qualifiche del programmatore.

L'indagine dell'agenzia federale Food and Drug Administration (FDA) 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 pensare: avete un piano formale per il collaudo dei vostri programmi?).

Le dosi eccessive 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 attività erano realizzate mediante una soluzione *ad hoc*, nonostante fossero già conosciute all'epoca 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 nel 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 disinformate 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? Sfortunatamente, ancora oggi non esistono standard aziendali che definiscono un processo sicuro per la progettazione del software.

## 7.8 Array a due dimensioni

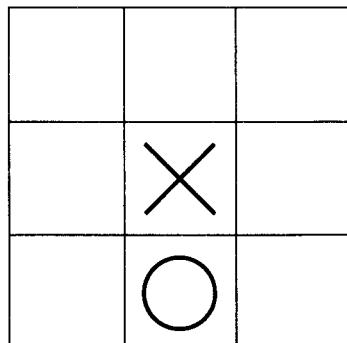
Gli array bidimensionali rappresentano una tabella, una disposizione di elementi in due dimensioni. Si accede agli elementi di un array bidimensionale usando una coppia di indici, `a[i][j]`.

Gli array e i vettori possono contenere sequenze lineari, ma a volte vi capiterà di voler memorizzare insiemi di dati che hanno una disposizione bidimensionale. Un tradizionale esempio è la scacchiera del gioco "Tic-Tac-Toe" (*tris o filetto*, Figura 15).

Una disposizione come questa, che consiste di righe e colonne di valori, viene definita *array bidimensionale* o *matrice*. Quando costruire un array bidimensionale, dovete specificare quante righe e quante colonne vi servono. In questo caso, vi servono 3 righe e 3 colonne:

**Figura 15**

Una scacchiera per il gioco  
"Tic-Tac-Toe" (tris o filetto)



```
final int ROWS = 3;
final int COLUMNS = 3;
String[][] board = new String[ROWS][COLUMNS];
```

Con questo si ottiene un array bidimensionale di 9 elementi:

```
board[0][0]    board[0][1]    board[0][2]
board[1][0]    board[1][1]    board[1][2]
board[2][0]    board[2][1]    board[2][2]
```

Per accedere a un particolare elemento della matrice, usate due indici fra parentesi quadre distinte:

```
board[1][1] = "x";
board[2][1] = "o";
```

Quando si riempie o si ispeziona un array bidimensionale, di solito si usano due cicli annidati. Ad esempio, questa coppia di cicli assegna a tutti gli elementi dell'array una stringa contenente il solo carattere di spaziatura.

```
for (int i = 0; i < ROWS; i++)
    for (int j = 0; j < COLUMNS; j++)
        board[i][j] = " ";
```

In questo ciclo abbiamo usato costanti per specificare il numero di righe e di colonne. Questi valori, che rappresentano le due dimensioni dell'array bidimensionale, possono, poi, essere ispezionati in questo modo:

- `board.length` è il numero di righe
- `board[0].length` è il numero di colonne (in Argomenti avanzati 7.3 troverete una spiegazione di questa espressione)

Il ciclo di riempimento della scacchiera può, quindi, essere riscritto così:

```
for (int i = 0; i < board.length; i++)
    for (int j = 0; j < board[0].length; j++)
        board[i][j] = " ";
```

Ecco una classe e un programma di prova per giocare a tic-tac-toe. Questa classe non controlla se un giocatore ha vinto la partita: come spesso si dice, “ciò viene lasciato al lettore come esercizio” (Esercizio P7.13).

### File ch07/twodim/TicTacToe.java

```
/**  
 * Una scacchiera 3x3 per il gioco tic-tac-toe.  
 */  
public class TicTacToe  
{  
    private String[][] board;  
    private static final int ROWS = 3;  
    private static final int COLUMNS = 3;  
  
    /**  
     * Costruisce una scacchiera vuota.  
     */  
    public TicTacToe()  
{  
        board = new String[ROWS][COLUMNS];  
        // riempì di spazi  
        for (int i = 0; i < ROWS; i++)  
            for (int j = 0; j < COLUMNS; j++)  
                board[i][j] = " ";  
    }  
    /**  
     * Imposta un settore della scacchiera.  
     * Il settore deve essere libero.  
     * @param i l'indice di riga  
     * @param j l'indice di colonna  
     * @param player il giocatore ("x" o "o")  
     */  
    public void set(int i, int j, String player)  
{  
        if (board[i][j].equals(" "))  
            board[i][j] = player;  
    }  
  
    /**  
     * Crea una rappresentazione della scacchiera  
     * sotto forma di stringa, come ad esempio  
     * | x o |  
     * | x |  
     * | o |.  
     * @return la stringa rappresentativa  
     */  
    public String toString()  
{  
        String r = "";  
        for (int i = 0; i < ROWS; i++)  
        {  
            r = r + "|";  
            for (int j = 0; j < COLUMNS; j++)  
                r = r + board[i][j];  
        }  
        return r;  
    }  
}
```

```
    r = r + "|\\n";
}
return r;
}
```

## File ch07/twodim/TicTacToeRunner.java

```
import java.util.Scanner;

/**
 * Questo programma esegue la classe TicTacToe
 * chiedendo all'utente di selezionare posizioni sulla
 * scacchiera e visualizzando il risultato.
 */
public class TicTacToeRunner
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        String player = "x";
        TicTacToe game = new TicTacToe();
        boolean done = false;
        while (!done)
        {
            System.out.println(game.toString());
            System.out.print("Row for " + player + " (-1 to exit): ");
            int row = in.nextInt();
            if (row < 0) done = true;
            else
            {
                System.out.print("Column for " + player + ": ");
                int column = in.nextInt();
                game.set(row, column, player);
                if (player.equals("x"))
                    player = "o";
                else
                    player = "x";
            }
        }
    }
}
```

## Esecuzione del programma

```
| |
| |
| |
| |
| |
Row for x (-1 to exit): 1
Column for x : 2
|
| |
| |
| |
| |
x
|
| |
| |
| |
| |
| |
Row for o (-1 to exit): 0
Column for o : 0
```

```

| 0 |
| x |
Row for x (-1 to exit): -1

```



## Auto-valutazione

19. Come si dichiara e si inizializza un array bidimensionale di interi con dimensione 4 per 4?
20. Come si conta il numero di caselle libere in una scacchiera per il gioco tic-tac-toe?



## Esempi completi 7.2

### Una tabella con la popolazione mondiale

Prendiamo in esame questi dati relativi alla popolazione mondiale:

| 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  |

Dovete progettare un programma che visualizzi questi dati sotto forma di tabella, aggiungendo il totale sotto ciascuna colonna, in modo da poter leggere, anno dopo anno, il valore della popolazione mondiale totale.

Per prima cosa, scomponiamo il problema in passi distinti:

Inizializza i dati della tabella.

Visualizza la tabella.

Calcola e visualizza i totali per colonna.

Inizializziamo la tabella come una sequenza di righe:

```

int[][] data =
{
    { 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 }
};

```

Per visualizzare le intestazioni delle righe, ci serve anche un'array unidimensionale con i nomi dei continenti, avente lo stesso numero di righe della tabella.

```
String[] continents =
{
    "Africa",
    "Asia",
    "Australia",
    "Europe",
    "North America",
    "South America"
}
```

Per visualizzare una riga, stampiamo prima il nome del continente, poi tutte le colonne. Per farlo usiamo due cicli annidati. Il ciclo più esterno visualizza ciascuna riga:

```
// visualizza i dati
for (int i = 0; i < ROWS; i++)
{
    // visualizza la riga i-esima
    ...
    System.out.println(); // va a capo alla fine della riga
}
```

Per visualizzare una riga, stampiamo prima la sua intestazione, poi tutte le colonne:

```
System.out.printf("%20s", continents[i]);
for (int j = 0; j < COLUMNS; j++)
{
    System.out.printf("%5d", data[i][j]);
}
```

Per visualizzare i totali per ciascuna colonna, usiamo l'algoritmo visto nel Paragrafo 7.6.2. facendo i calcoli colonna per colonna:

```
for (int j = 0; j < COLUMNS; j++)
{
    int total = 0;
    for (int i = 0; i < ROWS; i++)
    {
        total = total + data[i][j];
    }
    System.out.printf("%5d", total);
}
```

Ecco, infine, il programma completo.

```
public class WorldPopulation
{
    public static void main(String[] args)
    {
        final int ROWS = 6;
        final int COLUMNS = 7;

        int[][] data =
        {
```

```

    { 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 }
};

String[] continents =
{
    "Africa",
    "Asia",
    "Australia",
    "Europe",
    "North America",
    "South America"
}

System.out.println("      Year 1750 1800 1850 1900 1950 2000 2050");

// visualizza i dati
for (int i = 0; i < ROWS; i++)
{
    // visualizza la riga i-esima
    System.out.printf("%20s", continents[i]);
    for (int j = 0; j < COLUMNS; j++)
    {
        System.out.printf("%5d", data[i][j]);
    }
    System.out.println(); // va a capo alla fine della riga
}

// visualizza i totali in fondo a ciascuna colonna

System.out.println("          World");
for (int j = 0; j < COLUMNS; j++)
{
    int total = 0;
    for (int i = 0; i < ROWS; i++)
    {
        total = total + data[i][j];
    }
    System.out.printf("%5d", total);
}
System.out.println();
}
}

```

### Esecuzione del programma

|               | Year | 1750 | 1800 | 1850 | 1900 | 1950 | 2000 | 2050 |
|---------------|------|------|------|------|------|------|------|------|
| Africa        | 106  | 107  | 111  | 133  | 221  | 767  | 1766 |      |
| Asia          | 502  | 635  | 809  | 947  | 1407 | 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

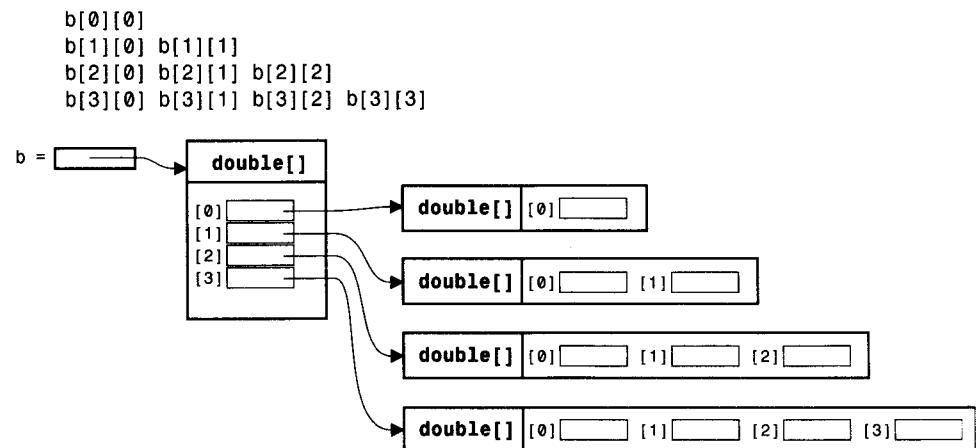
Quando dichiarate un array bidimensionale con l'enunciato

```
int[][] a = new int[3][3];
```

ottenete 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 sono della stessa lunghezza, ma in Java è possibile dichiarare anche array nei quali la lunghezza delle righe è variabile. Per esempio, potete creare un array che ha la forma di un triangolo, come questo:



Per creare un array come questo dovete faticare un po' di più. Per prima cosa create uno spazio adatto a contenere quattro righe, lasciando vuoto il secondo indice dell'array per indicare che imposterete manualmente ogni riga:

```
int[][] b = new int[4][];
```

Poi dovete creare ogni riga separatamente.

```
for (int i = 0; i < b.length; i++)
    b[i] = new int[i + 1];
```

Potete accedere a ciascun elemento dell'array usano la 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.

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();
}
```

Naturalmente, array come questi sono 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

Potete 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]
```

Questi array sono, però, veramente rari, soprattutto in programmi orientati agli oggetti, e non li prenderemo più in considerazione in questo libro.

## Riepilogo degli obiettivi di apprendimento

### Per memorizzare sequenze di valori si usano array

- Un array è una sequenza di valori del medesimo tipo, cioè omogenei.
- Si accede a un elemento di un array mediante un indice intero, usando l'operatore `[ ]`.
- In un array, i valori degli indici vanno da zero alla sua lunghezza, diminuita di uno.
- L'accesso a un elemento inesistente costituisce un errore di limiti.

- L'espressione `array.length` ha sempre un valore uguale al numero di elementi dell'`array`.
- Evitate di usare array paralleli, trasformandoli in array di oggetti.

### Per gestire sequenze di dimensione variabile si usano vettori

- La classe `ArrayList` gestisce una sequenza di oggetti; la lunghezza della sequenza è variabile.
- La classe `ArrayList` è una classe generica: `ArrayList<NomeTipo>` contiene oggetti di tipo `NomeTipo`.

### Le classi involucro consentono l'utilizzo di vettori di numeri

- Per manipolare valori di un tipo primitivo come se fossero oggetti si usano le classi involucro.

### Con il ciclo `for` esteso si scandiscono tutti gli elementi di una sequenza

- Il ciclo `for` esteso scandisce tutti gli elementi di una raccolta.
- In un ciclo `for` esteso la variabile di ciclo contiene un elemento, non un indice.

### Un array si può anche usare in modalità "riempito solo in parte"

- Insieme a un array riempito parzialmente, usate una variabile ausiliaria che tenga traccia del numero di elementi realmente utilizzati.

### È importante conoscere gli algoritmi di più frequente utilizzo con array/vettori

- Per contare i valori presenti in un vettore e aventi determinate caratteristiche, ispezionate tutti gli elementi e contate quelli che rispondono ai requisiti.
- Per trovare il valore massimo (o minimo), inizializzate un candidato con l'elemento iniziale, quindi confrontatelo con gli elementi rimanenti e aggiornatelo se trovate un elemento maggiore (o minore).
- Per trovare un valore, controllate tutti gli elementi finché non trovate quello cercato.
- Una variabile di tipo array memorizza un riferimento all'array. Copiando la variabile si ottiene un secondo riferimento al medesimo array.
- Per copiare gli elementi di un array, usate il metodo `Arrays.copyOf`.

### Occorre conoscere 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.

### Per dati organizzati in righe e colonne si usano array bidimensionali

- Gli array bidimensionali rappresentano una tabella, una disposizione di elementi in due dimensioni. Si accede agli elementi di un array bidimensionale usando una coppia di indici, `a[i][j]`.

## Classi, oggetti e metodi presentati nel capitolo

`java.lang.Boolean`  
    `booleanValue`  
`java.lang.Double`  
    `doubleValue`  
`java.lang.Integer`

`intValue`  
`java.util.Arrays`  
    `copyOf`  
    `toString`  
`java.util.ArrayList<E>`

|        |      |
|--------|------|
| add    | set  |
| get    | size |
| remove |      |

## Esercizi di ripasso

- ★ **Esercizio R7.1.** Cos'è un indice? Cosa sono e quali sono i limiti di un array e di un vettore? Cos'è un errore di limiti?
- ★ **Esercizio R7.2.** Scrivete un programma che contiene un errore di limiti ed eseguitelo. Cosa accade al vostro computer? Come venite aiutati dal messaggio d'errore a localizzare il problema?
- ★★ **Esercizio R7.3.** Scrivete il codice Java per un ciclo che calcola simultaneamente il valore massimo e il valore minimo presenti in un vettore. Usate come esempio un vettore di conti bancari.
- ★ **Esercizio R7.4.** Scrivete un ciclo che legge dieci stringhe e le inserisce in un vettore. Scrivete un secondo ciclo che stampa le stringhe in ordine inverso rispetto a come sono state inserite.
- ★★ **Esercizio R7.5.** Esamineate l'algoritmo che abbiamo utilizzato per identificare il valore massimo all'interno di un vettore: abbiamo assegnato il primo elemento a `largestYet` e, così facendo, non abbiamo più potuto utilizzare il ciclo `for` esteso. Un approccio alternativo consiste nell'assegnare il valore `null` a `largestYet`, per poi ispezionare tutti gli elementi: all'interno del ciclo occorre, naturalmente, verificare se `largestYet` abbia ancora il valore `null`. Modificate il ciclo che cerca il conto bancario avente saldo maggiore, utilizzando questa tecnica: questo approccio è più o meno efficiente di quello usato nel testo?
- ★★★ **Esercizio R7.6.** Considerate un'altra variante dell'algoritmo che identifica il valore massimo: in questo caso calcoliamo il valore massimo presente in un array di numeri.

```
double max = 0; // c'è un errore da qualche parte!
for (double element : values)
{
    if (element > max) max = element;
}
```

Questo approccio contiene, però, un errore piuttosto subdolo: qual è l'errore? Come lo si può correggere?

- ★ **Esercizio R7.7.** Per ciascuno dei seguenti insiemi di valori, scrivete un frammento di codice che li inserisca in un array `a`.
  - 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
  - 1 4 9 16 9 7 4 9 11

Quando è opportuno, usate un ciclo.

- ★★ **Esercizio R7.8.** Scrivete un ciclo che riempia un array `a` con dieci numeri casuali compresi fra 1 e 100. Scrivete un nuovo codice (usando uno o più cicli) che riempia `a` con dieci numeri casuali *diversi* compresi fra 1 e 100.

- \* **Esercizio R7.9.** Cosa c'è di sbagliato in questo ciclo?

```
double[] values = new double[10];
for (int i = 1; i <= 10; i++) values[i] = i * i;
```

Trovate due modi per correggere l'errore.

- \*\*\*\*T **Esercizio R7.10.** Scrivete un programma che costruisca un array di 20 interi e ne scriva i primi dieci elementi memorizzandovi i numeri 1, 4, 9, ..., 100. Compilate ed eseguite il debugger. Dopo che nell'array sono stati inseriti tre numeri, ispezionatelo. Cosa contengono gli elementi dell'array successivi a quelli che avete già scritto?

- \*\* **Esercizio R7.11.** Riscrivete i cicli seguenti senza usare il **for** esteso, tenendo presente che **values** è di tipo **double[]**.

- `for (double element : values) sum = sum + element;`
- `for (double element : values) if (element == target) return true;`
- `int i = 0;
 for (double element : values) { values[i] = 2 * element; i++; }`

- \*\* **Esercizio R7.12.** Riscrivete i cicli seguenti usando il **for** esteso, tenendo presente che **values** è di tipo **double[]**.

- `for (int i = 0; i < values.length; i++) sum = sum + values[i];`
- `for (int i = 1; i < values.length; i++) sum = sum + values[i];`
- `for (int i = 0; i < values.length; i++)
 if (values[i] == target) return i;`

- \*\* **Esercizio R7.13.** Cosa c'è di sbagliato in questo frammento di codice, che vuole stampare un vettore con separatori tra gli elementi?

```
System.out.print(values.get(0));
for (int i = 1; i < values.size(); i++)
{
    System.out.print(", " + values.get(i));
```

- \*\* **Esercizio R7.14.** Nel Paragrafo 7.6.6, per trovare la posizione di un valore che soddisfacesse una particolare condizione abbiamo usato un ciclo **while**, non un ciclo **for**. Cosa c'è di sbagliato in questo ciclo?

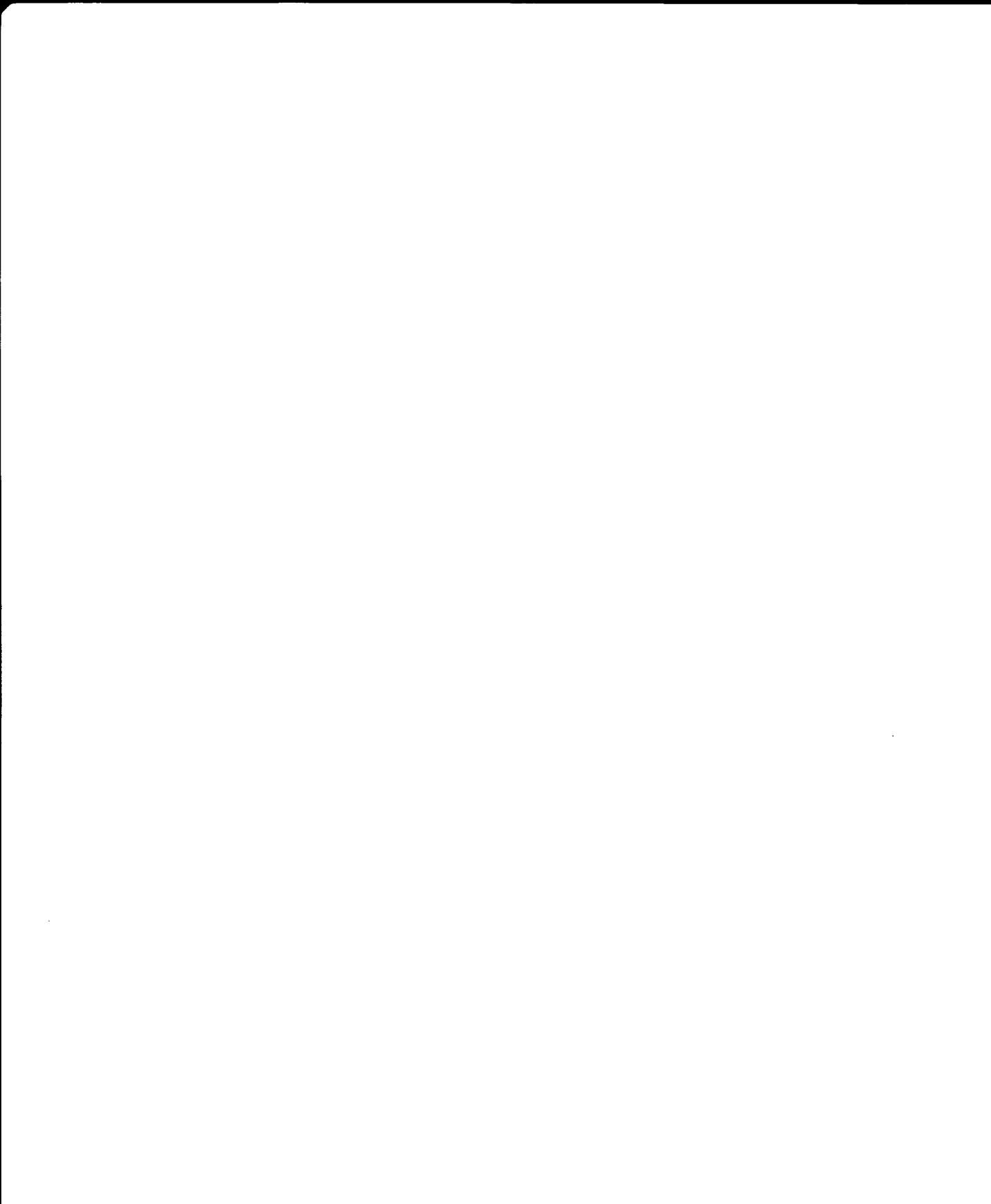
```
for (pos = 0; pos < values.size() && !found; pos++)
{
    if (values.get(pos) > 100)
    {
        found = true;
    }
}
```

- \*\* **Esercizio R7.15.** Nel Paragrafo 7.6.8, per inserire un elemento in una particolare posizione di un array abbiamo spostato tutti gli elementi a partire da tale posizione in avanti, iniziando dalla fine dell'array. Cosa c'è di sbagliato ad iniziare dalla posizione di inserimento, come in questo ciclo?

```
for (int i = pos; i < size - 1; i++)
{
    values[i + 1] = values[i];
}
```

- \*\* **Esercizio R7.16.** Nel Paragrafo 7.6.9, abbiamo raddoppiato la dimensione dell'array. Perché non abbiamo semplicemente aumentato la sua dimensione di un'unità?
- \* **Esercizio R7.17.** Cosa sono gli array paralleli? Perché gli array paralleli sono indicativi di una programmazione di scarso livello? Come si possono evitare?
- \* **Esercizio R7.18.** Vero o falso?
- Tutti gli elementi di un array sono dello stesso tipo.
  - Un indice di array deve essere un numero intero.
  - Gli array non possono avere come elementi riferimenti a stringhe.
  - Gli array non possono avere come elementi riferimenti `null`.
  - Gli array paralleli devono avere uguale lunghezza.
  - Gli array bidimensionali hanno sempre lo stesso numero di righe e di colonne.
  - Due array paralleli possono essere sostituiti da un array bidimensionale.
  - In un array bidimensionale gli elementi di colonne diverse possono avere tipi diversi.
- \*T **Esercizio R7.19.** Definite i termini *collaudo regressivo* e *pacchetto di prove* (“test suite”).
- \*T **Esercizio R7.20.** Nel collaudo, quale fenomeno viene detto “ciclicità”? Cosa si può fare per evitarlo?

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



# 8

## Progettazione di classi

### Obiettivi del capitolo

- Imparare a identificare le classi più appropriate da realizzare
- Assimilare i concetti di coesione e di accoppiamento
- Ridurre al minimo l'uso degli effetti collaterali
- Documentare con pre-condizioni e post-condizioni le responsabilità dei singoli metodi e di chi li invoca
- Apprendere l'utilizzo di metodi statici e variabili statiche
- Capire le regole di visibilità per variabili locali e variabili di esemplare
- Imparare l'utilizzo dei pacchetti
- Imparare a usare ambienti per il collaudo di unità

In questo capitolo imparerete altre cose sul progetto delle classi. Dapprima discuteremo il procedimento da usare per identificare le classi e dichiararne i metodi, poi vedremo come i concetti di pre-condizione e post-condizione vi consentano di specificare, realizzare e invocare i metodi in modo corretto. Imparerete anche alcuni dettagli più tecnici, come metodi statici e variabili statiche. Infine, vedrete come usare i pacchetti per organizzare le vostre classi.

## 8.1 Scegliere le classi

Nei capitoli precedenti avete usato un buon numero di classi e probabilmente avete anche progettato alcune vostre classi nello svolgimento degli esercizi di programmazione. 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
- Ellipse

mentre altre sono astrazioni di entità della vita reale

- BankAccount
- CashRegister

Le proprietà di un oggetto di queste classi sono facili da capire: un oggetto di tipo Rectangle ha una larghezza e un'altezza e un oggetto di tipo BankAccount consente di versare e prelevare denaro. In generale, concetti che appartengono all'ambito a cui si riferisce il vostro 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 esemplare 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 di tipo Scanner identifica numeri e stringhe in un flusso di caratteri, mentre un oggetto di tipo 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 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

**Una classe dovrebbe rappresentare un singolo concetto nel dominio in cui è descritto il problema: le scienze, la matematica o l'economia.**

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 “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: la parola “busta paga” è un sostantivo; potete immaginare un oggetto “busta paga”; potete, quindi, pensare a metodi utili della classe `Paycheck`, come `computeTaxes`, che vi aiutano 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”)?

## 8.2 Coesione e accoppiamento

In questo paragrafo imparerete due criteri utili per analizzare la qualità di una classe, cioè della sua interfaccia pubblica.

Una classe dovrebbe rappresentare un singolo concetto; i metodi pubblici e le costanti presenti nell'interfaccia pubblica dovrebbero avere una buona *coesione*, cioè tutte le caratteristiche dell'interfaccia dovrebbero essere strettamente correlate al singolo concetto rappresentato dalla classe.

Se scoprirete che l'interfaccia pubblica di una classe si riferisce a più di un concetto, forse è giunto il momento di usare classi separate. Considerate, ad esempio, l'interfaccia pubblica della classe `CashRegister` del Capitolo 4:

```
public class CashRegister
{
    public static final double NICKEL_VALUE = 0.05;
    public static final double DIME_VALUE = 0.1;
    public static final double QUARTER_VALUE = 0.25;
    ...
    public void enterPayment(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 P8.1 analizza una soluzione più generale).

Potrebbe essere più sensato disporre di una classe `Coin` 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 può, quindi, così semplificare:

```

public class CashRegister
{
...
public void enterPayment(int coinCount, Coin coinType) { ... }
...
}

```

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.

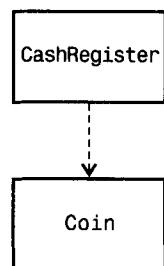
Molte classi hanno bisogno di altre classi per svolgere il proprio compito. Ad esempio, la classe `CashRegister` appena ristrutturata dipende ora dalla classe `Coin` per determinare il totale da pagare.

Per visualizzare relazioni come la dipendenza tra classi, i programmati tracciano 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 d’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`.

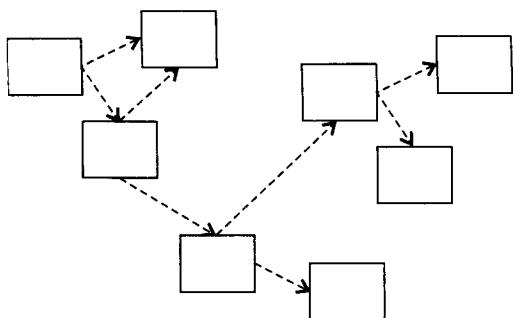
**Figura 1**

Relazione di dipendenza:  
la classe `CashRegister`  
dipende dalla classe `Coin`

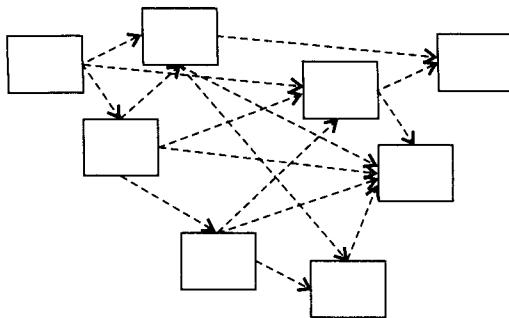


Notate che la classe `Coin` *non* dipende dalla classe `CashRegister`: le monete non sanno nemmeno di essere contenute in un registratore di cassa e svolgono i propri compiti senza mai invocare alcun metodo della classe `CashRegister`.

Una classe dipende da un'altra classe  
se usa suoi oggetti.



Basso accoppiamento



Elevato accoppiamento

**Figura 2**

Elevato o basso accoppiamento tra classi

*È buona norma rendere minimo l'accoppiamento (cioè le dipendenze) tra le classi.*

Se molte classi di un programma dipendono l'una dall'altra, diciamo che c'è un elevato *accoppiamento* tra le classi. Al contrario, se fra le classi esistono poche dipendenze, diciamo che l'accoppiamento è basso (osservate la Figura 2).

Perché l'accoppiamento è importante? Se la classe `Coin` viene modificata in una versione successiva del programma, tutte le classi che ne dipendono potrebbero aver bisogno di una modifica: se la modifica è drastica, tutte le classi accoppiate devono essere aggiornate. Inoltre, se volessimo usare la classe in un altro programma, dovremmo usare anche tutte le classi da cui dipende. Conseguentemente, vogliamo eliminare gli accoppiamenti non necessari tra le classi.



## 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 rendere minimo l'accoppiamento tra classi?



## Consigli per la qualità 8.1

### Coerenza

In questo paragrafo avete appreso due criteri per analizzare la qualità dell'interfaccia pubblica di una classe: dovreste massimizzare la coesione ed eliminare l'accoppiamento non indispensabile. 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 parametri: questo è, banalmente, un marchio di buona fattura.

Triste a dirsi, trovate 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 il parametro `null`? Il metodo `showMessageDialog` ha bisogno di un parametro che specifichi la finestra di appartenenza, parametro che deve valere `null` se non è necessaria una tale finestra. Perché questa incoerenza? Non c'è alcun motivo. Sarebbe stato semplice fornire un metodo `showMessageDialog` avente la stessa firma del metodo `showInputDialog`.

Incoerenze come questa non sono errori gravissimi, ma sono fastidiose, soprattutto perché si potrebbero evitare molto facilmente.

## 8.3 Classi immutabili

Quando analizziamo un programma costituito da molte classi, non è importante soltanto capire quali parti del programma usano una determinata classe, ma anche comprendere chi *modifica* oggetti di un certo tipo. Concentriamoci, qui, su questo aspetto della progettazione delle classi.

Come abbiamo già detto, un *metodo modificatore* modifica l'oggetto con il quale viene invocato, mentre un *metodo d'accesso* riporta semplicemente informazioni, senza effettuare alcuna modifica. Ad esempio, nella classe `BankAccount` i metodi `deposit` e `withdraw` sono modificatori, perché l'invocazione

```
account.deposit(1000);
```

modifica lo stato dell'oggetto `account`, mentre l'invocazione

```
double balance = account.getBalance();
```

non modifica lo stato di `account`.

**Una classe immutabile non ha metodi modificatori.**

Potete invocare un metodo d'accesso tutte le volte che volete: riceverete sempre la stessa risposta e lo stato dell'oggetto non cambierà. Si tratta, chiaramente, di una proprietà allietante, che rende ben prevedibile il comportamento di un metodo del genere.

Alcune classi, dette *immutabili*, sono progettate per 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:

```
String name = "John Q. Public";
String uppercased = name.toUpperCase(); // 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 modo inaspettato. Al contrario, se concedete a un metodo qualunque un riferimento a un oggetto di tipo `BankAccount`, dovete essere consapevoli che lo stato del vostro oggetto può cambiare: il metodo può invocare `deposit` e `withdraw` usando il riferimento che avete fornito.



## Auto-valutazione

6. Il metodo `substring` della classe `String` è un metodo d'accesso o un metodo modificatore?
7. La classe `Rectangle` è immutabile?

## 8.4 Effetti collaterali

Un effetto collaterale di un metodo è una qualsiasi modifica apportata ai dati che sia osservabile all'esterno del metodo.

Un *effetto collaterale* di un metodo è qualunque tipo di comportamento che sia osservabile al di fuori del metodo stesso: i metodi modificatori hanno un effetto collaterale, perché modificano il proprio parametro implicito. Ad esempio, dopo aver invocato

```
harrysChecking.deposit(1000);
```

potete osservare, invocando `harrysChecking.getBalance()`, che qualcosa è cambiato.

Vediamo ora possibili effetti sul parametro esplicito di un metodo, come `studentNames` in questo codice:

```
public class GradeBook
{
    ...
    /**
     * Aggiunge nomi di studenti a questo registro scolastico.
     * @param studentNames un elenco di nomi di studenti
     */
    public void addStudents(ArrayList<String> studentNames)
    {
        while (studentNames.size() > 0)
        {
            String name = studentNames.remove(0); // sconsigliato
            aggiungi name al registro
        }
    }
}
```

Mentre li inserisce nel registro scolastico, questo metodo *elimina* tutti i nomi dal vettore a cui fa riferimento il parametro `studentNames`: anche questo è un effetto collaterale. Infatti, dopo aver invocato

```
book.addStudents(listOfNames);
```

l'invocazione di `listOfNames.size()` restituisce 0. Un effetto collaterale come questo risulterebbe probabilmente inatteso per la maggior parte dei programmati: è molto meglio che il metodo legga i nomi dall'elenco senza modificarlo.

Consideriamo ora questo metodo:

```
public class BankAccount
{
    ...
}
```

```

    /**
     * Trasferisce denaro da questo conto a un altro.
     * @param amount la somma di denaro da trasferire
     * @param other il conto nel quale si trasferisce denaro
    */
    public void transfer(double amount, BankAccount other)
    {
        balance = balance - amount;
        other.deposit(amount);
    }
}

```

Questo metodo modifica sia il parametro implicito sia il parametro esplicito `other`. In questo caso non c'è alcun motivo per cercare di evitare questi effetti collaterali: trattandosi di un metodo che si chiama `transfer`, nessuno si meraviglierà.

Un altro esempio di effetto collaterale è la visualizzazione di dati in uscita. Osserviamo come abbiamo sempre stampato il saldo di un conto bancario:

```
System.out.println("The balance is now $" + momSavings.getBalance());
```

Perché non abbiamo semplicemente progettato un metodo `printBalance`?

```

public void printBalance() // sconsigliato
{
    System.out.println("The balance is now $" + balance);
}

```

Sarebbe una soluzione più comoda quando volete effettivamente visualizzare il valore del saldo, ma, naturalmente, esistono casi in cui il saldo vi serve per altri motivi, per cui non potete semplicemente eliminare il metodo `getBalance` sostituendolo con `printBalance`.

Una considerazione ancora più importante: il metodo `printBalance` fa forti ipotesi sulla classe `BankAccount`.

- Il messaggio è in inglese: avete ipotizzato che l'utente del vostro software sia in grado di leggere e capire la lingua inglese, ma la maggioranza della popolazione del pianeta non ha questa capacità.
- Usate l'oggetto `System.out`: un metodo che usa l'oggetto `System.out` non funzionerà in un sistema “embedded”, come il piccolo calcolatore che si trova all'interno di una segreteria telefonica.

In altre parole, un metodo con un effetto collaterale viola la regola che tende a minimizzare l'accoppiamento tra le classi: il metodo `printBalance` accoppia la classe `BankAccount` con le classi `System` e `PrintStream`. È meglio tenere distinte le attività di ingresso e uscita di dati dal vero compito assegnato alla vostra classe.



## Auto-valutazione

- Se `a` è un riferimento a un conto bancario, allora l'invocazione `a.deposit(100)` modifica l'oggetto di tipo conto bancario. Si tratta di un effetto collaterale?

9. Considerate la classe `Dataset` del Capitolo 6. Se vi aggiungessimo questo metodo, avrebbe un effetto collaterale diverso dalla sola modifica dell'insieme di dati?

```
void read(Scanner in)
{
    while (in.hasNextDouble())
        add(in.nextDouble());
}
```

## Errori comuni 8.1

### Tentare di modificare parametri di tipo primitivo

I metodi non possono modificare i parametri di tipo primitivo (numeri, `char` e `boolean`). Per illustrare meglio questo fatto, cerchiamo di scrivere un metodo che aggiorni un parametro numerico:

```
public class BankAccount
{
    ...
    /**
     * Trasferisce denaro da questo conto a un altro.
     * @param amount la somma di denaro da trasferire
     * @param otherBalance il saldo a cui aggiungere la somma trasferita
    */
    public void transfer(double amount, double otherBalance)
    {
        balance = balance - amount;
        otherBalance = otherBalance + amount; // non funzionerà
    }
}
```

Questo non funzionerà. Consideriamo un'invocazione del metodo:

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance);
System.out.println(savingsBalance);
```

Come si vede nella Figura 3, quando inizia l'esecuzione del metodo, la variabile parametro `otherBalance` assume lo stesso valore di `savingsBalance`. Viene poi modificato il valore di `otherBalance`, ma tale modifica non ha alcun effetto su `savingsBalance`, perché `otherBalance` è una variabile diversa. Quando il metodo termina, la variabile `otherBalance` non esiste più e `savingsBalance` non è stata incrementata.

Perché funziona l'esempio presentato all'inizio del Paragrafo 8.4, dove il secondo parametro esplicito era un riferimento di tipo `BankAccount`? In quel caso la variabile parametro conteneva una copia del riferimento all'oggetto: attraverso tale riferimento, il metodo è in grado di modificare l'oggetto.

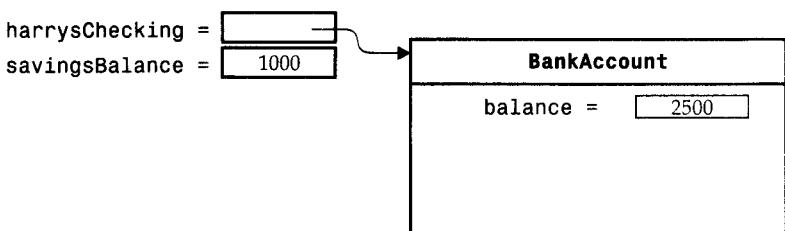
Avevi già visto, nel Capitolo 2, questa differenza fra oggetti e tipi primitivi. Conseguentemente, un metodo Java non può mai modificare numeri che gli vengono passati come parametri.

In Java, un metodo non può mai modificare propri parametri di tipo primitivo.

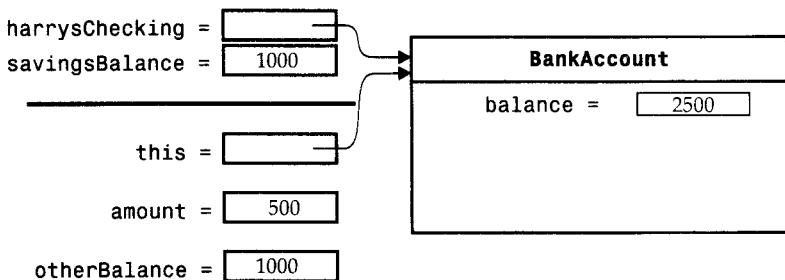
**Figura 3**

La modifica di un parametro numerico non è visibile da chi ha invocato il metodo

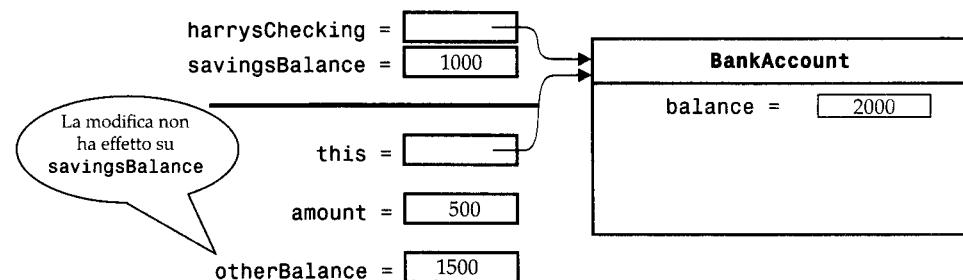
① Prima dell'invocazione del metodo



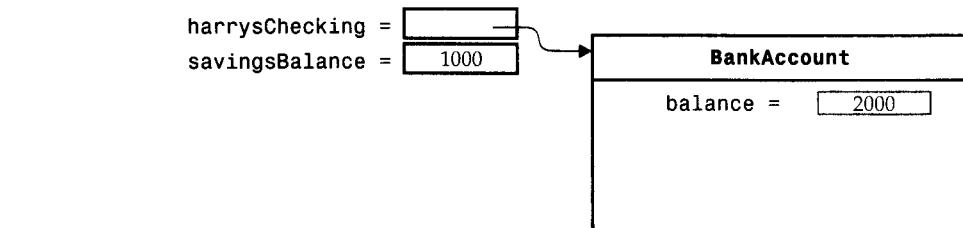
② Inizializzazione dei parametri del metodo



③ Subito prima di terminare l'esecuzione del metodo



④ Dopo l'invocazione del metodo



## Consigli per la qualità 8.2

### Ridurre al minimo gli effetti collaterali

In un mondo ideale, tutti i metodi sarebbero metodi d'accesso, che si limitano a restituire una risposta senza modificare alcun valore (e, in effetti, i programmi che sono scritti nei cosiddetti linguaggi di programmazione *funzionali*, come Scheme e ML, si avvicinano a

questo ideale). Naturalmente, in un linguaggio di programmazione orientato agli oggetti, usiamo gli oggetti per memorizzare cambiamenti di stato, pertanto, un metodo che modifica soltanto lo stato del suo parametro implicito è certamente accettabile. Sebbene gli effetti collaterali non si possano eliminare completamente, possono causare stupore e problemi e bisogna ridurli al minimo.

Nella progettazione di metodi,  
occorre minimizzare i loro effetti  
collaterali.

Analizzando gli effetti collaterali, possiamo catalogare i metodi in questo modo:

- Metodi d'accesso che non producono cambiamenti su eventuali parametri esplicativi: nessun effetto collaterale. Esempio: `getBalance`.
- Metodi modificatori che non producono cambiamenti su eventuali parametri esplicativi: un effetto collaterale accettabile. Esempio: `withdraw`.
- Metodi che modificano un parametro esplicito: un effetto collaterale che dovrebbe essere evitato ogni volta che ciò è possibile. Esempio: `transfer` è accettabile, ma `addStudents` non lo è.
- Metodi che modificano un altro oggetto (come `System.out`): un effetto collaterale che dovrebbe essere evitato. Esempio: `printBalance`.

## Consigli per la qualità 8.3

### Non modificate il contenuto di variabili parametro

Come detto in Errori comuni 8.1 e Argomenti avanzati 8.1, un metodo può trattare le proprie variabili parametro come qualsiasi altra variabile locale e modificarne il contenuto. Tuttavia, tale modifica ha effetto soltanto sulla variabile parametro all'interno del metodo stesso, non sui valori forniti dal metodo invocante. Alcuni programmati "sfruttano" la natura temporanea delle variabili parametro e le usano come "comodi" contenitori per risultati intermedi, come in questo esempio:

```
public void deposit(double amount)
{
    // si usa la variabile parametro per memorizzare un valore intermedio
    amount = balance + amount; // pessimo stile
    ...
}
```

Questo codice produrrebbe errori se all'interno del metodo un altro enunciato facesse riferimento alla variabile `amount` pensando che abbia il valore del parametro; inoltre, ciò confonderà i programmati che faranno la manutenzione di questo metodo. Dovreste sempre trattare le variabili parametro come se fossero costanti: non assegnate loro alcun valore. Piuttosto, definite una nuova variabile locale.

```
public void deposit(double amount)
{
    double newBalance = balance + amount;
    ...
}
```



## Argomenti avanzati 8.1

### Invocazione per valore e invocazione per riferimento

In Java, quando un metodo inizia la propria esecuzione, le sue variabili parametro vengono inizializzate copiandovi i valori forniti nell'invocazione del metodo stesso. Gli informatici chiamano questo meccanismo, che presenta alcune limitazioni, invocazione (o chiamata "per valore"). Come avete visto in Errori comuni 8.1, non è possibile realizzare metodi che modifichino il contenuto di variabili numeriche. Altri linguaggi di programmazione, come C++, consentono l'utilizzo di un meccanismo alternativo, l'invocazione "per riferimento". Per esempio, in C++ è semplice scrivere un metodo che modifichi un numero, usando un cosiddetto *parametro di tipo riferimento*. Ecco il codice C++, per chi di voi lo conosce:

```
// questo è codice C++
class BankAccount
{
public:
    void transfer(double amount, double& otherBalance)
        // otherBalance è di tipo double&, cioè è un riferimento a double
    {
        balance = balance - amount;
        otherBalance = otherBalance + amount; // in C++ funziona
    }
    ...
};
```

Nei manuali Java, talvolta 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. Per vederlo chiaramente, esaminiamo un altro scenario. Questo metodo tenta di assegnare al parametro **otherAccount** un riferimento a un nuovo oggetto:

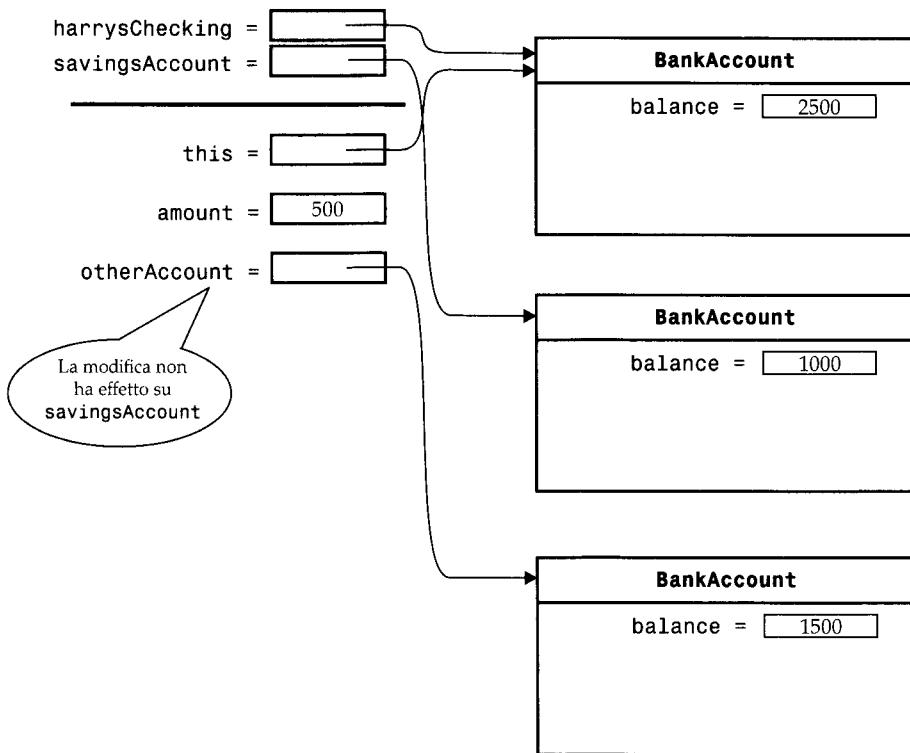
```
public class BankAccount
{
    ...
    public void transfer(double amount, BankAccount otherAccount)
    {
        balance = balance - amount;
        double newBalance = otherAccount.balance + amount;
        otherAccount = new BankAccount(newBalance); // non funziona
    }
}
```

**In Java, un metodo può modificare lo stato dell'oggetto a cui si riferisce uno dei suoi parametri, ma non può sostituire tale riferimento con un altro.**

Qui non stiamo tentando di cambiare lo stato dell'oggetto a cui si riferisce la variabile parametro **otherAccount**: stiamo tentando di sostituire l'oggetto con un oggetto diverso. Ora, la variabile parametro **otherAccount** viene sostituita con un riferimento a un nuovo conto bancario, ma, se invocate il metodo in questo modo

```
harrysChecking.transfer(500, savingsAccount);
```

La modifica di un parametro di tipo riferimento non è visibile da chi ha invocato il metodo



la modifica non ha effetto sulla variabile `savingsAccount` usata nell'invocazione.

Come potete vedere, in Java un metodo può aggiornare lo stato di un oggetto, ma non può *sostituire* il contenuto di un riferimento a oggetto. Questo dimostra che, in Java, anche i riferimenti a oggetto vengono passati per valore.

## 8.5 Pre-condizioni e post-condizioni

Una pre-condizione è un requisito che deve essere soddisfatto da chi invoca un metodo.

Una *pre-condizione* è un requisito che deve essere soddisfatto da chi invoca un metodo. Ad esempio, il metodo `deposit` della classe `BankAccount` prevede, come pre-condizione, che la somma da versare non sia negativa. Chi invoca un metodo ha la responsabilità di farlo rispettando tutte le relative pre-condizioni: se un metodo viene invocato in violazione di una delle proprie pre-condizioni, non è tenuto a produrre un risultato corretto.

Di conseguenza, una pre-condizione è parte essenziale del metodo e dovete documentarla. Ecco la documentazione della pre-condizione relativa al parametro `amount`, che non deve essere negativo.

```

/**
 * Versa denaro in questo conto bancario.
 * @param amount la somma di denaro da versare
 * (Pre-condizione: amount >= 0)
 */

```

Alcune estensioni di javadoc elaborano i marcatori `@precondition` o `@requires`, ma ciò non fa parte del programma javadoc ordinario. Poiché il programma javadoc non considera e non legge eventuali marcatori sconosciuti, aggiungiamo semplicemente la pre-condizione alla spiegazione del metodo o al marcatore `@param` appropriato.

Le pre-condizioni vengono tipicamente enunciate per uno di questi due motivi:

1. per restringere il campo di variabilità dei parametri di un metodo;
2. per richiedere che un metodo venga invocato soltanto quando l'oggetto su cui agisce si trova in uno *stato* appropriato.

Ad esempio, una volta che un oggetto di tipo `Scanner` ha esaurito i dati in ingresso, non è più lecito invocare il suo metodo `next`: il fatto che `hasNext` restituisca `true` è, quindi, una pre-condizione per il metodo `next`.

Un metodo è tenuto a operare correttamente soltanto quando tutte le sue pre-condizioni sono state soddisfatte, altrimenti è libero di fare *qualsiasi cosa*. Cosa dovrebbe fare, in pratica, un metodo invocato con dati inappropriate? Per esempio, che effetto si dovrebbe provocare scrivendo `account.deposit(-1000)`? Ci sono due scelte valide.

1. Un metodo può accettare la violazione e *lanciare un'eccezione*. In questo modo il metodo non fa proseguire l'esecuzione del metodo chiamante, ma, invece, trasferisce il controllo a un gestore di eccezione. Se non è presente un gestore, il programma si arresta. Parleremo delle eccezioni nel Capitolo 11.
2. Un metodo può semplicemente portare a compimento l'elaborazione assumendo che le proprie pre-condizioni siano soddisfatte: se non lo sono, qualsiasi dato errato (come, ad esempio, un saldo negativo) o altri errori saranno addebitabili al chiamante.

Il primo approccio può essere poco efficiente, in modo particolare se più metodi eseguono molte volte la medesima verifica, mentre il secondo approccio può essere pericoloso. Per ottenere i vantaggi di entrambi gli approcci è stato inventato il meccanismo delle *asserzioni*.

Un'*asserzione* è una condizione che dovrebbe essere sempre vera in un determinato punto del programma. La verifica di un'*asserzione* controlla che l'*asserzione* sia vera. Ecco una tipica asserzione, utilizzata per verificare una pre-condizione:

```
public void deposit(double amount)
{
    assert amount >= 0;
    balance = balance + amount;
}
```

In questo metodo il programmatore si aspetta che la quantità `amount` non sia mai negativa. Quando tale asserzione è vera non succede nulla e il programma funziona normalmente. Se, per qualche motivo, l'*asserzione* non è vera ed è stata abilitata la verifica delle *asserzioni*, il programma termina con la segnalazione di un `AssertionError`.

Se, invece, durante l'esecuzione la verifica delle *asserzioni* è stata disabilitata, allora l'*asserzione* non viene mai controllata e il programma viene eseguito senza rallentamenti; tale condizione è assicurata nella normale esecuzione dei programmi Java, salvo diversa e specifica opzione che abiliti la verifica delle *asserzioni*, in questo modo:

**Se un metodo viene invocato in violazione di una sua pre-condizione, non è tenuto a produrre un risultato corretto.**

**Un'asserzione è una condizione logica che si ipotizza essere vera all'interno di un programma.**

```
java -enableassertions ClasseEseguibile
```

Al posto di `-enableassertions` si può usare l'abbreviazione `-ea`. Durante la fase di sviluppo e collaudo di un programma, sarà assolutamente opportuno abilitare la verifica delle asserzioni.

Non è strettamente necessario utilizzare le asserzioni per la verifica delle pre-condizioni: si ritiene accettabile anche il lancio di un'eccezione, ma le asserzioni hanno il vantaggio di poter essere facilmente disabilitate dopo il collaudo del programma, che potrà così essere eseguito, in seguito, alla velocità massima. In questo modo non dovete mai preoccuparvi per l'inserimento di troppe asserzioni nel vostro codice: potete usarle per verificare anche altre condizioni, oltre alle pre-condizioni.

Molti programmatore inesperti ritengono che non sia “bello” provocare la terminazione prematura di un programma dopo aver accertato la violazione di una pre-condizione: perché non restituire semplicemente il controllo al chiamante?

```
public void deposit(double amount)
{
    if (amount < 0)
        return; // sconsigliato
    balance = balance + amount;
}
```

Questo è lecito: dopo tutto, se le proprie pre-condizioni non sono soddisfatte, un metodo può legittimamente decidere di non far nulla. Questa soluzione, però, non è efficace quanto la verifica di un'asserzione. Se il programma che invoca il metodo `deposit` ha alcuni errori che provocano l'utilizzo di una somma di denaro negativa come parametro fornito al metodo, la versione che genera un errore nella verifica di asserzione renderà molto evidenti tali errori durante il collaudo: è difficile non accorgersi di una brusca terminazione del programma. La versione silenziosa, invece, non vi mette in allarme e potreste non notare che, di conseguenza, esegue in modo errato alcuni calcoli. Pensate alle asserzioni come a un approccio di tipo “rimprovero amorevole” alla verifica delle pre-condizioni.

## Sintassi di Java

### 8.1 Afferzione

#### Sintassi

```
assert condizione;
```

#### Esempio

```
assert amount >= 0;
```

Se la condizione è falsa e la verifica delle asserzioni è abilitata, viene lanciata un'eccezione.

Condizione che deve essere vera.

**Se un metodo viene invocato nel rispetto delle proprie pre-condizioni, deve garantire che le relative post-condizioni siano valide.**

Un metodo che venga invocato nel rispetto delle proprie pre-condizioni promette di svolgere correttamente il proprio compito e fa anche promesse di tipo diverso, chiamate *post-condizioni*, che ricadono in due categorie:

1. che il valore di ritorno venga calcolato correttamente;
2. che l'oggetto su cui agisce il metodo si trovi in un determinato stato dopo che l'esecuzione del metodo è terminata.

Ecco una post-condizione che fa un'affermazione a proposito dello stato dell'oggetto dopo che il metodo `deposit` è stato eseguito.

```
/**
 * Versa denaro in questo conto bancario.
 * (Post-condizione: getBalance() >= 0)
 * @param amount la somma di denaro da versare
 * (Pre-condizione: amount >= 0)
 */
```

Se la pre-condizione è soddisfatta, questo metodo garantisce che, dopo il versamento, il saldo non sia negativo.

Alcune estensioni di javadoc elaborano i marcatori `@postcondition` o `@ensures`, ma ciò non fa parte del programma javadoc ordinario. Poiché il programma javadoc non considera e non legge eventuali marcatori sconosciuti, aggiungiamo semplicemente le post-condizioni alla spiegazione del metodo o al marcitore `@return`, come già abbiamo fatto con le pre-condizioni.

Alcuni programmatore pensano di dover specificare una post-condizione in ogni metodo, ma, quando usate javadoc, specificate già nel marcitore `@return` una parte di post-condizione, che non dovreste ripetere.

```
// Questo enunciato di post-condizione è oltremodo ripetitivo
/**
 * Restituisce il saldo attuale di questo conto bancario.
 * @return il saldo attuale
 * (Post-condizione: il valore restituito è uguale al saldo del conto)
 */
```

Note che abbiamo formulato le pre-condizioni e le post-condizioni soltanto in termini di *interfaccia pubblica* della classe, per cui enunciamo, ad esempio, la pre-condizione del metodo `withdraw` come `amount <= getBalance()`, e non `amount <= balance`. Dopo tutto, chi invoca il metodo e deve soddisfare la pre-condizione, ha accesso solamente all'interfaccia pubblica, non all'implementazione privata.

Pre-condizioni e post-condizioni sono spesso paragonate ai contratti, che, nella vita reale, specificano le condizioni che devono essere rispettate dai contraenti. Ad esempio, il vostro meccanico può promettervi di riparare la vostra automobile e voi promettete, per contro, di pagare una determinata somma di denaro: se uno dei contraenti rompe la promessa, l'altro non è tenuto al rispetto dei termini contrattuali. Allo stesso modo, pre-condizioni e post-condizioni sono termini contrattuali fra un metodo e chi lo invoca. Il metodo si impegna a soddisfare le post-condizioni ogniqualvolta i dati in ingresso soddisfino le pre-condizioni, mentre il chiamante promette di non invocare il metodo con

dati proibiti. Se l'invocante onora la propria promessa, ma riceve una risposta sbagliata, può citare in giudizio il metodo di fronte al “tribunale dei programmatori”. Se, invece, l'invocante non rispetta i propri impegni e, di conseguenza, succede qualcosa di terribile, non può obiettare.

## Auto-valutazione

10. Per quale motivo si aggiunge una pre-condizione a un metodo che viene reso disponibile ad altri programmatori?
11. Quando progettate un metodo che ha una pre-condizione e vi accorgete che l'invocante non l'ha soddisfatta, lo dovete avvertire?

## Argomenti avanzati 8.2

### Invarianti di classe

Nella sezione Argomenti avanzati 6.5 avete visto il concetto di *condizione invariante in un ciclo*. Una condizione invariante in un ciclo è valida prima che il ciclo venga eseguito per la prima volta e si mantiene valida dopo ogni iterazione del ciclo. Sappiamo, quindi, che la condizione invariante di un ciclo deve essere valida anche dopo la conclusione definitiva del ciclo e possiamo usare tale informazione per dimostrare la correttezza del ciclo stesso.

Una condizione invariante di classe serve a uno scopo simile: è un'affermazione, relativa a un oggetto, che è vera dopo l'esecuzione di uno qualsiasi dei costruttori e che continua a essere vera dopo l'esecuzione di qualsiasi metodo modificatore (a patto che l'invocazione del metodo rispetti tutte le relative pre-condizioni). Sappiamo, quindi, che una condizione invariante di classe deve sempre essere vera e possiamo usare tale informazione per dimostrare la correttezza del nostro programma.

Ecco un semplice esempio. Consideriamo la classe `BankAccount`, specificando le seguenti pre-condizioni per il costruttore e per i metodi modificatori:

```
public class BankAccount
{
    ...
    /**
     * Costruisce un conto bancario con un saldo assegnato.
     * @param initialBalance il saldo iniziale
     * (Pre-condizione: initialBalance >= 0)
    */
    public BankAccount(double initialBalance)
    {
        balance = initialBalance;
    }

    /**
     * Versa una somma di denaro nel conto bancario.
     * @param amount la somma da versare
     * (Pre-condizione: amount >= 0)
    */
    public void deposit(double amount) { ... }
```

```

    /**
     * Preleva una somma di denaro dal conto bancario.
     * @param amount la somma da prelevare
     * (Pre-condizione: amount <= getBalance())
    */
    public void withdraw(double amount) { ... }
}

```

A questo punto possiamo formulare la seguente condizione invariante di classe:

```
getBalance() >= 0
```

Per verificare se questa condizione invariante è vera, controlliamo prima il costruttore. Poiché la pre-condizione del costruttore è

```
initialBalance >= 0
```

possiamo dimostrare che l'invariante è vera dopo che il costruttore ha impostato `balance` al valore `initialBalance`.

Controlliamo ora i metodi modificatori. La pre-condizione del metodo `deposit` è

```
amount >= 0
```

Possiamo ipotizzare che la condizione invariante sia vera prima di invocare il metodo, per cui sappiamo che `balance >= 0` prima dell'esecuzione del metodo stesso. Le regole della matematica ci dicono che la somma di due numeri non negativi è ancora un numero non negativo, per cui possiamo concludere che `balance >= 0` anche dopo l'esecuzione di un versamento. Quindi, il metodo `deposit` preserva l'invariante.

Un ragionamento simile dimostra che anche il metodo `withdraw` preserva l'invariante.

Poiché l'invariante è una proprietà della classe, la documenterete nella descrizione della classe:

```

    /**
     * Un conto bancario ha un saldo che può essere
     * modificato con versamenti e prelievi.
     * (Invariante: getBalance() >= 0)
    */
    public class BankAccount
    {
        ...
    }

```

## 8.6 Metodi statici

Un metodo statico non viene invocato mediante un oggetto.

Talvolta servono metodi che non vengono invocati con un oggetto come parametro隐式: un metodo di questo tipo viene detto *metodo di classe* o *metodo statico* (dal nome della parola riservata, `static`, che si usa per dichiararlo). Per contro, i metodi che avete visto nei paragrafi precedenti spesso sono chiamati *metodi di esemplare* (o di istanza), perché operano su un particolare esemplare di un oggetto.

Un tipico esempio di metodo statico è il metodo `sqrt` della classe `Math`. Quando scrivete `Math.sqrt(x)`, non specificate alcun parametro implicito (ricordatevi che `Math` è il nome di una classe, non è un oggetto).

Per quale ragione si vuole scrivere un metodo che non opera su un oggetto? Il motivo più diffuso è che si vogliono incapsulare alcune elaborazioni che coinvolgono solamente numeri: dal momento che i numeri non sono oggetti, non potete utilizzarli per invocare metodi. Ad esempio, l'invocazione `x.sqrt()` sarebbe illegale in Java.

Ecco un tipico esempio di metodo statico che esegue un semplice calcolo algebrico, la percentuale `p` della quantità `a`. Poiché i parametri sono numeri, il metodo non opera assolutamente su alcun oggetto, per cui lo possiamo progettare sotto forma di metodo statico:

```
/*
    Calcola la percentuale di una data quantità.
    @param p la percentuale da applicare
    @param a la quantità di cui calcolare la percentuale
    @return la percentuale p di a
*/
public static double percentOf(double p, double a)
{
    return (p / 100) * a;
}
```

**Quando si progetta un metodo statico, occorre individuare una classe in cui inserirlo.**

Dobbiamo ora capire dove poter scrivere questo metodo. Inventiamoci una nuova classe, simile alla classe `Math` della libreria standard di Java: dal momento che questo metodo ha a che fare con calcoli finanziari, progetteremo una classe `Financial` per contenerlo. Ecco la classe:

```
public class Financial
{
    public static double percentOf(double p, double a)
    {
        return (p / 100) * a;
    }
    // qui si possono aggiungere altri metodi finanziari
}
```

Quando invocate un metodo statico, scrivete il nome della classe che lo contiene, in modo che il compilatore lo possa trovare, come in questo esempio:

```
double tax = Financial.percentOf(taxRate, total);
```

Notate che per invocare questo metodo non si usa un oggetto di tipo `Financial`.

C'è anche un altro motivo che, a volte, rende necessari i metodi statici: se un metodo elabora una classe di cui non siete i progettisti, non lo potete aggiungere a quella classe. Pensate a un metodo che calcoli l'area di un rettangolo: la classe `Rectangle` fa parte della libreria standard e non dispone di tale funzionalità, però non la possiamo modificare. Allora un metodo statico risolve il nostro problema:

```
public class Geometry
{
    public static double area(Rectangle rect)
```

```

    {
        return rect.getWidth() * rect.getHeight();
    }
    // qui si possono aggiungere altri metodi geometrici
}

```

A questo punto possiamo finalmente spiegare perché il metodo `main` è statico: nel momento in cui un programma viene eseguito, non esiste ancora alcun oggetto, pertanto il *primo* metodo del programma deve essere necessariamente un metodo statico.

È lecito domandarsi perché questi metodi siano detti *static*, dal momento che il normale significato della parola *statico* ("che sta fermo in un posto") non sembra avere nulla in comune con ciò che fanno i metodi statici. Senza dubbio è vero, con la "scusante" che Java utilizza la parola chiave `static` perché il linguaggio C++ la impiega nello stesso contesto. Poi, va detto che C++ usa `static` per indicare i metodi di classe perché... i progettisti del linguaggio non vollero inventarsi un'altra parola riservata: qualcuno osservò che esisteva una parola riservata usata piuttosto raramente, `static`, per indicare variabili che rimangono in una posizione fissa durante più invocazioni successive di metodi (Java non ha questa caratteristica e nemmeno ne ha bisogno). Pertanto, fu possibile riutilizzare la stessa parola chiave per indicare i metodi di classe senza confondere il compilatore, mentre il fatto che potesse confondere gli esseri umani non sembrò un grosso problema. Quindi, dovrete proprio convivere con il fatto che "metodo statico" significa, in realtà, "metodo di classe", vale a dire un metodo che ha soltanto parametri esplicativi.



## Auto-valutazione

12. Immaginate che Java non consentisse la definizione di metodi statici. Come usereste, in tal caso, il metodo `sqrt` della classe `Math` per calcolare la radice quadrata del numero `x`?
13. Perché questo metodo, che calcola la media dei numeri presenti in un vettore, deve essere statico?  
`public static double average(ArrayList<Double> values)`



## Consigli per la qualità 8.4

### Limitare al massimo 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 nello sviluppo 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 aver eliminato il voto minimo. Abbiamo risolto il problema realizzando una classe, `GradeBook`, 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 1;
    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 `GradeBook` in modo che memorizzi i voti di molti studenti, mentre aggiungere funzionalità a metodi statici genera rapidamente confusione (come si può vedere nell'Esercizio P8.7).

## 8.7 Variabili statiche

A volte occorre memorizzare valori che appartengono più correttamente a una classe che a un suo oggetto: a questo scopo si utilizza una *variabile statica*. 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 conservare da qualche parte l'ultimo numero di conto assegnato.

Non ha senso, però, memorizzare questo valore in una variabile di esemplare:

```

public class BankAccount
{
    private double balance;
    private int accountNumber;
    private int lastAssignedNumber = 1000; // NO, non funziona
    ...
}

```

In questo caso, ciascun *esemplare* della classe `BankAccount` avrebbe il proprio valore di `lastAssignedNumber`.

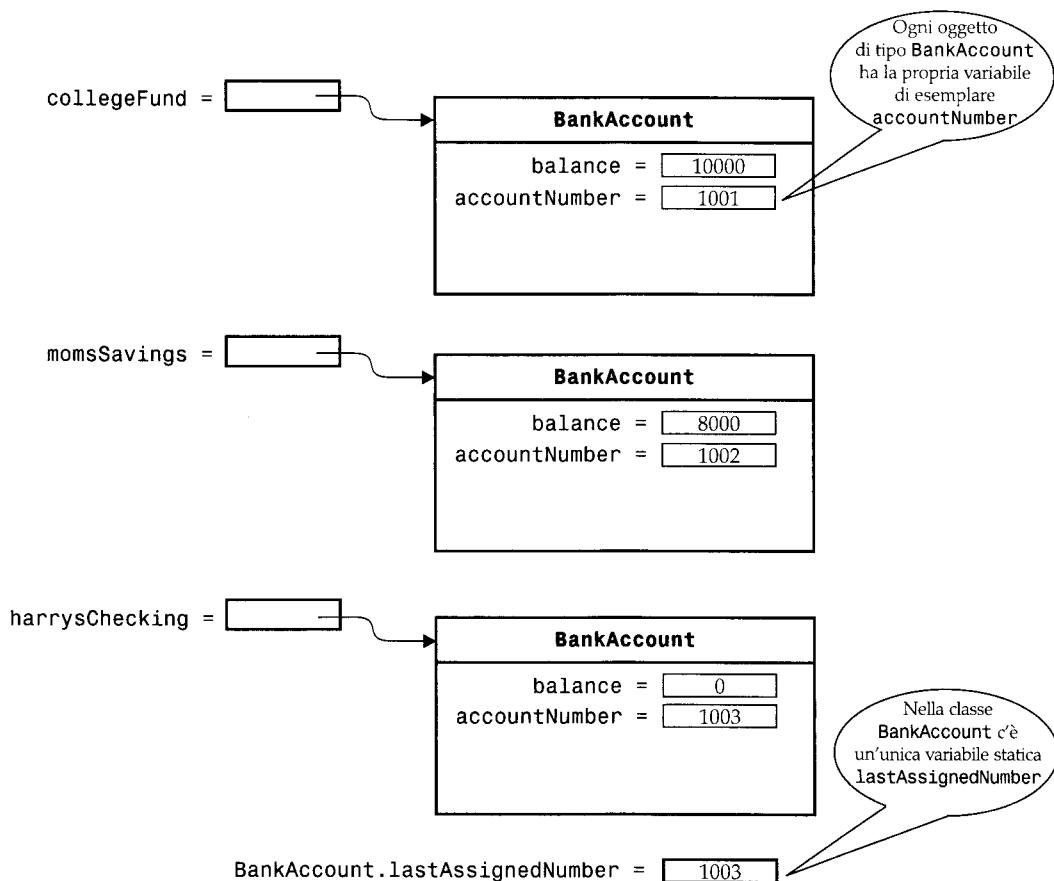
Al contrario, a noi serve memorizzare un unico valore in `lastAssignedNumber`: l'ultimo numero assegnato, lo stesso per tutta la *classe*. Una variabile di questo tipo è detta *statica* e si dichiara mediante la parola riservata `static`.

```

public class BankAccount
{
    private double balance;
    private int accountNumber;
    private static int lastAssignedNumber = 1000;
    ...
}

```

Una variabile statica appartiene alla classe, non a un oggetto della classe.

**Figura 4**

Ciascun oggetto di tipo `BankAccount` ha le proprie variabili di esemplare `balance` e `accountNumber`, però esiste un'unica copia della variabile `lastAssignedNumber` (osservate la Figura 4), collocata in una porzione di memoria a sé stante, al di fuori di qualsiasi oggetto di tipo `BankAccount`.

Una variabile statica viene a volte chiamata anche *variabile di classe*, proprio perché ne esiste un'unica copia per l'intera classe.

Ciascun metodo o costruttore di una classe può accedere alle variabili statiche della classe stessa. Ecco il costruttore della classe `BankAccount`, che incrementa l'ultimo numero assegnato per poi utilizzarlo per inizializzare il numero di conto dell'oggetto che sta costruendo:

```

public class BankAccount
{
    ...
    public BankAccount()
    {
        lastAssignedNumber++; // aggiorna la variabile di classe
        accountNumber = lastAssignedNumber; // aggiorna la variabile di esemplare
    }
}
  
```

Esistono tre modi per inizializzare una variabile statica:

1. Non fare nulla. La variabile statica viene inizializzata automaticamente a zero (per i numeri), a `false` (per i valori `boolean`) oppure a `null` (per gli oggetti).
2. Usare un inizializzatore esplicito, come questo:

```
public class BankAccount
{
    private static int lastAssignedNumber = 1000;
    ...
}
```

3. Usare un blocco di inizializzazione statica (descritto nella sezione Argomenti avanzati 8.4).

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`.

Dichiarare `static` le costanti è molto sensato: non è di nessuna utilità che ogni oggetto della classe abbia un proprio insieme di variabili con valori costanti, è sufficiente un solo insieme per l'intera classe.

Perché queste variabili di classe si chiamano “statiche”? Come per i metodi “statici”, la parola riservata `static` è solo un’eredità priva di significato, che proviene dal C++.  
Però le variabili statiche e i metodi statici hanno molto in comune: si applicano a tutta la *classe*, non a sue istanze specifiche.

In generale si vuole minimizzare l’utilizzo di metodi statici e di variabili statiche. Se vi accorgete che state utilizzando molti metodi statici che usano variabili statiche, è probabile che non abbiate identificato le classi migliori per la soluzione del vostro problema secondo un approccio orientato agli oggetti.

## Auto-valutazione

14. Citate due variabili statiche della classe `System`.
15. Un amico vi dice che ha trovato un’ottima strategia per evitare l’utilizzo di quegli antipatici oggetti: mettere tutto il codice in un’unica classe e dichiarare 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?





## Argomenti avanzati 8.3

### Importazione statica

A partire dalla versione 5.0 del linguaggio Java, 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(PI)
        out.println(x); // invece di System.out
    }
}
```

Le importazioni statiche rendono i programmi di più agevole lettura, in particolare quando si utilizzano molte funzioni matematiche.



## Argomenti avanzati 8.4

### Forme alternative per inizializzare variabili di esemplare e variabili statiche

Come avete visto, le variabili di esemplare vengono inizializzate mediante un valore predefinito (`0`, `false` o `null`, in relazione al tipo). In seguito, secondo la procedura che preferiamo in questo libro, potete assegnare loro qualsiasi valore desiderato, 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 predefiniti 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 quando si costruisce un oggetto. Ecco un esempio:

```
public class Coin
{
    private double value;
    private String name;
    {
```

```

        value = 1;
        name = "Dollar";
    }
}

```

Per le variabili statiche, usate 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.

## 8.8 Ambito di visibilità

L'ambito di visibilità di una variabile è la porzione di programma da cui si può accedere alla variabile.

L'*ambito di visibilità* (in inglese, *scope*) di una variabile è la porzione di programma da cui si può accedere alla variabile. Rendere minimo l'ambito di visibilità di una variabile è una buona pratica di programmazione, perché riduce la possibilità di modifiche accidentali e di conflitto fra nomi.

In questo paragrafo imparerete a determinare l'ambito di visibilità delle variabili locali e di esemplare, oltre a capire come vengono risolti i conflitti tra nomi quando gli ambiti si sovrappongono.

### 8.8.1 Visibilità di variabili

L'ambito di visibilità di una variabile locale si estende dal punto in cui viene dichiarata fino alla fine del blocco o del ciclo `for` che la contiene, mentre l'ambito di visibilità di una variabile parametro è l'intero metodo.

```

public static void process(double[] values) // values è una variabile
  // parametro
{
    for (int i = 0; i < 20; i++) // i è una variabile locale dichiarata
                                  // in un ciclo for
    {
        if (values[i] == 0)

```

```

{
    double r = Math.random(); // r è una variabile locale dichiarata
                            // in un blocco
    values[i] = r;
} // qui termina la visibilità di r
} // qui termina la visibilità di i
} // qui termina la visibilità di values

```

L'ambito di visibilità di una variabile locale non può contenere la definizione di un'altra variabile locale avente lo stesso nome.

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. Questo, ad esempio, è un errore:

```

public static void main(String[] args)
{
    double r = Math.random();
    if (r > 0.5)
    {
        Rectangle r = new Rectangle(5, 10, 20, 30);
        // ERRORE: qui non si può dichiarare un'altra variabile locale di nome r
        ...
    }
}

```

Se, però, i loro ambiti non si sovrappongono, potete avere variabili locali con nomi identici, come in questo esempio:

```

if (Math.random() > 0.5)
{
    Rectangle r = new Rectangle(5, 10, 20, 30);
    ...
} // la visibilità di r termina qui
else
{
    int r = 5;
    // va bene, qui si può dichiarare un'altra variabile r
    ...
}

```

### 8.8.2 Visibilità sovrapposte

Se esistono due variabili con lo stesso nome e con ambiti sovrapposti sorgono problemi. Ciò non può mai accadere con variabili locali, ma gli ambiti di visibilità di una variabile locale e di una variabile di esemplare possono sovrapporsi. Ecco un esempio di codice in cui ciò accade: sintatticamente valido, ma stilisticamente pessimo.

```

public class Coin
{
    private String name;
    private double value; // variabile di esemplare
    ...
    public double getExchangeValue(double exchangeRate)
    {
        double value; // variabile locale con lo stesso nome
        ...
    }
}

```

```
        return value;  
    }  
}
```

All'interno del metodo `getExchangeValue`, il nome di variabile `value` potrebbe avere due significati: quello della **variabile locale** e quello della **variabile di esemplare**. Il linguaggio Java specifica che, in questa situazione, prevale il nome della variabile `locale`, che *mette in ombra* la variabile di esemplare.

Sembra una decisione piuttosto arbitraria, ma si basa su un valido motivo: potete sempre riferirvi alla variabile di esemplare scrivendo `this.value`.

```
value = this.value * exchangeRate;
```

Scrivere codice di questo genere non è una buona idea: potete cambiare facilmente il nome della variabile locale in qualcos'altro, come `result`.

C'è però un utilizzo piuttosto comune di questa sovrapposizione tra ambiti di visibilità. Nella realizzazione di costruttori o di metodi che impostano il valore di una variabile, è noioso escogitare nomi diversi per i parametri e per le corrispondenti variabili di esempio. Usando `this` si può usare lo stesso nome:

```
public Coin(double value, String name)
{
    this.value = value;
    this.name = name;
}
```

L'espressione `this.value` fa riferimento alla variabile di esemplare, mentre `value` è il parametro.



## **Auto-valutazione**

16. Questo programma usa due variabili di nome r. È sintatticamente corretto?

```
public class RectangleTester
{
    public static double area(Rectangle rect)
    {
        double r = rect.getWidth() * rect.getHeight();
        return r;
    }

    public static void main(String[] args)
    {
        Rectangle r = new Rectangle(5, 10, 20, 30);
        double a = area(r);
        System.out.println(r);
        ...
    }
}
```

17. Qual è l'ambito di visibilità della variabile di esemplare `balance` della classe `BankAccount`?



## Errori comuni 8.2

### Mettere in ombra le variabili (*shadowing*)

Utilizzare accidentalmente lo stesso nome per una variabile locale e per una variabile di esemplare è un errore sorprendentemente comune. Come avete appena visto, la variabile locale *mette in ombra* la variabile di esemplare e, nonostante l'eventuale intenzione di accedere alla variabile di esemplare, si accede erroneamente alla variabile locale senza nessuna segnalazione da parte del compilatore. Osservate questo esempio di un costruttore errato:

```
public class Coin
{
    private double value;
    private String name;
    ...
    public Coin(double aValue, String aName)
    {
        value = aValue;
        String name = aName; // ah! ...
    }
}
```

Il programmatore ha dichiarato una variabile locale `name` nel costruttore. Molto probabilmente si è trattato di un errore di battitura: le dita del programmatore viaggiavano con il pilota automatico e hanno scritto la parola `String`, sebbene l'autore avesse la ferma intenzione di accedere alla variabile di esemplare. Sfortunatamente, in queste situazioni il compilatore non segnala nulla e imposta tranquillamente la variabile locale al valore di `aName`: quindi, la variabile di esemplare dell'oggetto che si sta costruendo non viene mai modificata e rimane al valore `null`.

Alcuni programmatore danno a tutte le variabili di esemplare nomi che hanno un prefisso speciale, per distinguerle dalle altre variabili. Una convenzione diffusa è l'utilizzo del prefisso `my`, come `myValue` o `myName`.

Un altro modo per evitare questo problema consiste nell'utilizzo del parametro `this` per effettuare tutti gli accessi a variabili di esemplare, in questo modo:

```
this.name = name;
```



## Consigli per la qualità 8.5

### Minimizzare l'ambito di visibilità di ciascuna variabile

Se fate in modo che l'ambito di visibilità di una variabile sia minimo, le erronee modifiche accidentali divengono meno probabili e, durante eventuali ristrutturazioni del codice, risulta più facile modificare o eliminare la variabile stessa.

Come già detto, non rendete pubblica una variabile di esemplare. Nella libreria di Java, alcune classi hanno variabili di esemplare pubbliche, ma i loro progettisti hanno poi rimpianto le loro decisioni, quando si sono resi conto di non poter più modificare e migliorare quel codice.

Ogni variabile dovrebbe avere l'ambito di visibilità più ridotto possibile.

Quando dichiarate una costante, chiedetevi chi può averne bisogno. Tutti (`public static final`)? Soltanto chi progetta la classe (`private static final`)? Oppure un solo metodo (una variabile locale `final`)? Scegliete l'ambito più ristretto.

Fate molta attenzione alle variabili di esemplare inutili. Ad esempio, prendiamo in esame la classe `Pyramid` vista negli Esempi completi 4.1, nella quale non vogliamo usare una variabile di esemplare per memorizzare il volume:

```
public class Pyramid
{
    private double height;
    private double baseLength;
    private double volume;
    // usare l'ambito di visibilità di esemplare per questa variabile
    // non è una buona idea
    ...
}
```

Piuttosto, calcoliamo il volume quando serve, nel metodo `getVolume`: in questo modo, nessun altro metodo potrà modificare accidentalmente la variabile `volume` né, invece, dimenticarsi di modificarla quando la base o l'altezza cambiano.

Infine, diciamo una parola sulle variabili locali: dichiaratele soltanto nel punto in cui vi servono.

## 8.9 Pacchetti

Un pacchetto (*package*) è un insieme di classi correlate.

Un programma Java è, in generale, costituito da più classi. Finora, 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 costituiscono questo strumento 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 importanti della libreria di Java

| Pacchetto                | Scopo                                                | Esempio di classi        |
|--------------------------|------------------------------------------------------|--------------------------|
| <code>java.lang</code>   | Supporto al linguaggio                               | <code>Math</code>        |
| <code>java.util</code>   | Varie utilità                                        | <code>Random</code>      |
| <code>java.io</code>     | Input e output                                       | <code>PrintStream</code> |
| <code>java.awt</code>    | Abstract Windowing Toolkit                           | <code>Color</code>       |
| <code>java.applet</code> | Applet                                               | <code>Applet</code>      |
| <code>java.net</code>    | Connessione di rete                                  | <code>Socket</code>      |
| <code>java.sql</code>    | Accesso a database tramite Structured Query Language | <code>ResultSet</code>   |
| <code>javax.swing</code> | Interfaccia utente Swing                             | <code>JButton</code>     |
| <code>omg.w3c.dom</code> | Document Object Model (DOM) per documenti XML        | <code>Document</code>    |

**Sintassi di Java****8.2 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.

**8.9.1 Organizzare classi in pacchetti**

Per inserire una delle vostre classi in un pacchetto, dovete scrivere questa riga come prima istruzione del file sorgente che contiene la classe:

```
package nomePacchetto;
```

Come potete vedere dagli esempi nella libreria di Java, il nome di un pacchetto è formato da uno o più identificatori, separati da punti (nel Paragrafo 8.9.3 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* o *pacchetto di default*, 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.

**8.9.2 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.

Naturalmente, questo modo di scrivere il codice è molto scomodo. In alternativa, potete *importare* un nome mediante un enunciato `import`:

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

Successivamente, 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 termina con `.*`. Per esempio, potete usare l'enunciato seguente per importare tutte le classi del pacchetto `java.util`:

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

Questo enunciato vi 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`.

### 8.9.3 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 *confitto di nomi*. In un progetto di grandi dimensioni, è inevitabile che due persone escogitino 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 siano univoci. Sarebbe inopportuno che il costruttore di auto BMW inserisse tutto il suo codice Java nel pacchetto `bmw`, e che qualche altro programmatore (magari Britney M. Walters) avesse la stessa brillante idea. Per evitare questo problema, gli inventori di Java raccomandano di utilizzare uno schema per l'assegnazione dei nomi ai pacchetti che sfrutta l'univocità dei nomi dei domini in Internet.

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 le componenti del nome del dominio, creando un prefisso per i nomi dei pacchetti, come `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

Per costruire nomi di pacchetti non ambigui usate un nome di dominio, invertendone le componenti.

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 esercitatori 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.

#### 8.9.4 Come vengono localizzate le classi

Il percorso del file che contiene una classe deve corrispondere al suo nome di pacchetto.

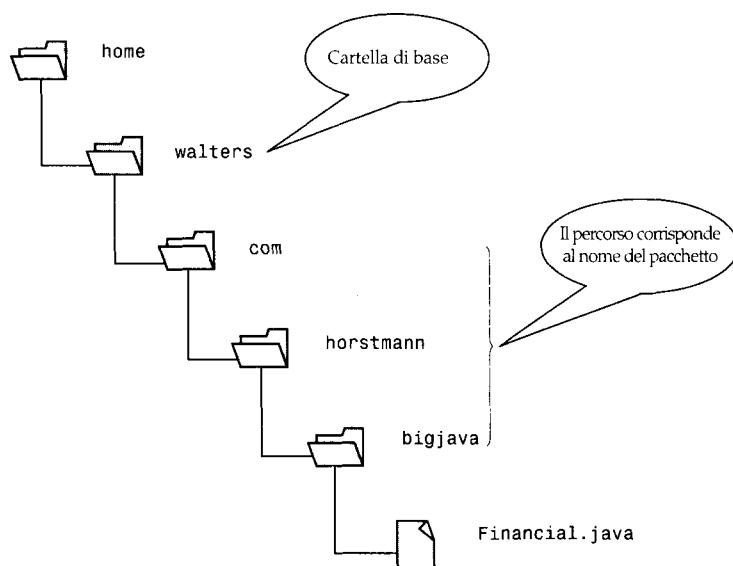
Un file sorgente, contenente una classe, deve trovarsi in una cartella il cui nome corrisponde a quello del pacchetto a cui appartiene la classe. Le parti del nome del pacchetto, separate da punti, rappresentano le cartelle annidate, una dentro l'altra. Per esempio, le classi del pacchetto `com.horstmann.bigjava` devono essere nella cartella `com/horstmann/bigjava` all'interno della *cartella di base* ("base directory") in cui svolgete le vostre esercitazioni. Se, ad esempio, la vostra cartella di base è `/home/walters`, potete collocare i file delle classi del pacchetto `com.horstmann.bigjava` nella cartella `/home/walters/com/horstmann/bigjava`, come mostrato nella Figura 5 (stiamo usando nomi di file secondo lo stile di UNIX; in Windows, usereste `c:\home\walters\com\horstmann\bigjava`).



#### Auto-valutazione

18. Quali di questi nomi rappresentano pacchetti?
  - a. `java`
  - b. `java.lang`
  - c. `java.util`
  - d. `java.lang.Math`
19. Un programma Java privo di enunciati `import` è vincolato a usare soltanto classi appartenente al pacchetto predefinito o al pacchetto `java.lang`?

**Figura 5**  
Cartella di base e cartelle per i pacchetti



20. Se per i vostri programmi usate la cartella di base `/home/me/cs101` (probabilmente `c:\Users\Me\cs101` in Windows) e il vostro insegnante vi chiede di inserire in pacchetti le vostre soluzioni degli esercizi del testo, in quale cartella metterete la classe `hw1.problem1.TicTacToeTester`?

## Erri<sup>o</sup> comuni 8.3

### 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 un oggetto o una classe che abbia un metodo statico `println` (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 di esemplare `out`, oppure una classe con una variabile `static`. 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 codifica rigoroso. Se i nomi delle classi 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 8.5

### 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. 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.

Variabili e metodi che non vengono dichiarati né `public` né `private` sono disponibili per tutte le classi dello stesso pacchetto, situazione solitamente indesiderata.

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. Ad esempio, può darsi che il vostro istruttore vi chieda di inserire ciascuna vostra esercitazione in un pacchetto separato: in tal modo, in pacchetti distinti potete avere classi con lo stesso nome ma con implementazioni diverse (come `homework1.problem1.Bank` e `homework1.problem2.Bank`).

#### Fase 1 Scegliete un nome di pacchetto

Se il vostro istruttore 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 sottopacchetto che descriva il vostro progetto o la vostra esercitazione, come `edu.sjsu.cs.walters.cs1project`.

#### 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 la vostra 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 *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 Java, 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 nel nome del file. Ad esempio

```
cd /home/britney  
java homework1.problem2.Tester
```

oppure

```
c:  
cd \Users\Britney  
java homework1.problem2.Tester
```



## Note di cronaca 8.1

### Lo sviluppo esplosivo 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 potersi connettere 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 simili, e occorsero anni perché l'industria informatica si accorgesse che un'autentica unità di elaborazione centrale 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 per la prima volta, 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. L'originale 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 manager intermedi delle società, che apprezzavano i calcolatori ma erano stanchi di aspettare ore o giorni perché i loro dati tornassero dal centro di calcolo, si affrettarono a comprare VisiCalc e il computer che serviva per eseguirlo: per loro, il computer era una macchina per il foglio elettronico.

Il successivo grosso passo in avanti fu il Personal Computer di IBM, che soltanto in seguito divenne noto come "PC". Era il primo computer per uso personale a grande diffusione che usava il processore Intel 8086 a 16 bit, i cui discendenti attualmente vengono ancora usati nei personal computer. Il successo del PC era dovuto non a qualche innovazione nel progetto, ma al fatto che fosse facile da *clonare*. IBM pubblicò le specifiche per le schede di espansione e fornì anche un aiuto ulteriore: diffuse l'esatto codice sorgente del cosid-

detto BIOS (Basic Input/Output System, sistema di base per la gestione dei flussi in ingresso e in uscita), che serve per controllare la tastiera, lo schermo, le porte e le unità disco, e che bisogna installare in ciascun PC, nella memoria ROM. Ciò permise ai fornitori di schede di espansione di garantire che il codice del BIOS e le sue estensioni prodotte da terzi interagissero correttamente con l'apparecchiatura. Naturalmente, il codice era di proprietà di IBM e, quindi, non si poteva copiare legalmente: forse la società non aveva previsto che altri potessero, nonostante questo, ricreare versioni del BIOS funzionalmente equivalenti. Compaq, uno dei primi produttori di cloni, si avvalse del lavoro di quindici ingegneri, che certificarono di non avere mai visto il codice originale IBM e che scrissero una nuova versione di BIOS che si atteneva esattamente alle specifiche IBM. Altre aziende fecero lo stesso e presto vi fu una grande quantità di produttori di PC, sui quali veniva eseguito lo stesso software del PC IBM, ma che si distinguevano per il prezzo basso o per le prestazioni migliori. Con il tempo, IBM perse la sua posizione dominante nel mercato dei PC e, nel 2005, ha ceduto la propria linea di personal computer al produttore cinese Lenovo.

IBM non ha mai prodotto per i suoi PC un *sistema operativo*, il software che organizza l'interazione fra l'utente e il computer, esegue i programmi applicativi e gestisce l'accesso al disco e alle altre risorse: IBM offriva ai propri clienti la scelta fra tre sistemi operativi diversi. Per

La maggioranza dei clienti la scelta del sistema operativo non era assolutamente importante e sceglievano quello che fosse in grado di eseguire la maggior parte delle poche applicazioni che esistevano a quel tempo. La scelta cadde su DOS (Disk Operating System) prodotto da Microsoft, che concesse liberamente l'utilizzo dello stesso sistema operativo anche agli altri produttori di hardware e incoraggiò le società di software a scrivere applicazioni per DOS. Il risultato fu un numero enorme di utili programmi applicativi per macchine compatibili con il PC.

Le applicazioni per PC erano certamente utili, ma non erano facili da imparare. Ciascun produttore sviluppò una diversa *interfaccia utente*, ossia un insieme di combinazioni di tasti, di opzioni di menu e di impostazioni che un utente doveva padroneggiare per usare un programma in modo efficace. Lo scambio di dati fra applicazioni era difficoltoso, perché ciascun programma utilizzava un formato diverso per i dati. Nel 1984, il computer Macintosh di Apple diede una svolta a tutto questo: i suoi progettisti riuscirono a dotarlo di un'interfaccia utente intuitiva, con-

vincendo gli sviluppatori di software ad adeguarsi. A Microsoft e ai costruttori di PC compatibili occorsero anni per recuperare.

Oggi la maggior parte dei personal computer viene utilizzata per accedere a informazioni presenti in Internet, per l'intrattenimento, per l'elaborazione di testi e per la contabilità domestica (operazioni bancarie, bilancio familiare e fiscale). Alcuni analisti prevedono che il personal computer si fonderà con la televisione via cavo, per diventare un *elettrodomestico per le informazioni e l'intrattenimento*.

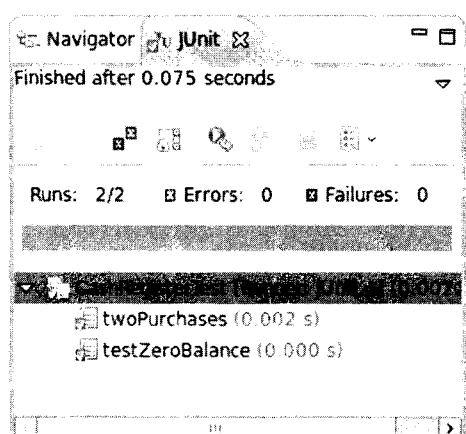
## 8.10 Ambienti per il collaudo di unità

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, 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 e valutare velocemente pacchetti di prove, oltre a rendere più semplice l'aggiunta di casi di prova a pacchetti 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 6**  
Collaudo di unità con JUnit



Lavorando con JUnit, per ogni classe che viene sviluppata si progetta una classe ausiliaria per il collaudo, con un metodo per ogni caso di prova che volete 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 accettabile per i valori in virgola mobile.

C'è anche la consuetudine (non necessaria) che il nome della classe di collaudo termini con `Test`, come `CashRegisterTest`. Esaminate questo esempio:

```
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.enterPayment(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 di non introdurre più il medesimo errore 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

21. Scrivete con JUnit una classe di prova con un caso di prova per la classe `Earthquake` vista nel Capitolo 5.
22. Che significato ha il parametro `EPSILON` nel metodo `assertEquals`?

## 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.

### Analizzare la coesione e l'accoppiamento di classi

- 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 classe se usa suoi oggetti.
- È buona norma rendere minimo l'accoppiamento (cioè le dipendenze) tra le classi.

### Riconoscere le classi immutabili e sfruttarne i vantaggi

- Una classe immutabile non ha metodi modificatori.
- I riferimenti a oggetti di una classe immutabile possono essere condivisi in sicurezza.

### Riconoscere gli effetti collaterali e capire che vanno minimizzati

- Un effetto collaterale di un metodo è una qualsiasi modifica apportata ai dati che sia osservabile all'esterno del metodo.
- In Java, un metodo non può mai modificare propri parametri di tipo primitivo.
- Nella progettazione di metodi, occorre minimizzare i loro effetti collaterali.
- In Java, un metodo può modificare lo stato dell'oggetto a cui si riferisce uno dei suoi parametri, ma non può sostituire tale riferimento con un altro.

### Documentare pre-condizioni e post-condizioni dei metodi

- Una pre-condizione è un requisito che deve essere soddisfatto da chi invoca un metodo.
- Se un metodo viene invocato in violazione di una sua pre-condizione, non è tenuto a produrre un risultato corretto.
- Un'asserzione è una condizione logica che si ipotizza essere vera all'interno di un programma.
- Se un metodo viene invocato nel rispetto delle proprie pre-condizioni, deve garantire che le relative post-condizioni siano valide.

### Saper realizzare metodi statici, che non operano su oggetti

- Un metodo statico non viene invocato mediante un oggetto.
- Quando si progetta un metodo statico, occorre individuare una classe in cui inserirlo.

### Usare variabili statiche per descrivere le proprietà di una classe

- Una variabile statica appartiene alla classe, non a un oggetto della classe.

### Determinare l'ambito di visibilità di variabili locali e di esemplare

- L'ambito di visibilità di una variabile è la porzione di programma da cui si può accedere alla variabile.

- L'ambito di visibilità di una variabile locale non può contenere la definizione di un'altra variabile locale avente lo stesso nome.
- Una variabile locale può mettere in ombra una variabile di esemplare avente lo stesso nome. Si può accedere alla variabile messa in ombra usando il riferimento `this`.
- Ogni variabile dovrebbe avere l'ambito di visibilità più ridotto possibile.

### 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, invertendone le componenti.
- 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 ripasso

**★★ Esercizio R8.1.** Quali classi usereste per risolvere il problema così descritto?

Gli utenti inseriscono monete in un distributore automatico e selezionano un prodotto premendo un pulsante. Se le monete inserite sono sufficienti a raggiungere il prezzo di acquisto del prodotto, questo viene fornito insieme al resto, altrimenti, le monete inserite vengono restituite all'utente.

**★★ Esercizio R8.2.** Quali classi usereste per risolvere il problema così descritto?

Gli impiegati ricevono la loro busta paga ogni due settimane e vengono pagati con la loro paga oraria per ogni ora lavorata. Se, però, hanno lavorato più di 40 ore in una settimana, le ore di straordinario vengono pagate il 150% della paga ordinaria.

**★★ Esercizio R8.3.** Quali classi usereste per risolvere il problema così descritto?

I clienti ordinano prodotti in un negozio. Vengono generate fatture che elencano gli oggetti e le quantità ordinate, i pagamenti ricevuti e le somme ancora dovute. I prodotti vengono inviati all'indirizzo di destinazione del cliente, mentre le fatture vengono spedite all'indirizzo di fatturazione.

**★★ Esercizio R8.4.** Osservate l'interfaccia pubblica della classe `java.lang.System` e analizzate la sua coesione.

**★★ Esercizio R8.5.** 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`, `Customer` e `Product`.

**★★ Esercizio R8.6.** Supponete che un distributore automatico contenga prodotti e che gli utenti vi inseriscano monete per comprarli. Disegnate un diagramma UML che mostri le dipendenze tra le classi `VendingMachine`, `Coin` e `Product`.

- \*\* **Esercizio R8.7.** Da quali classi dipende la classe `Integer` della libreria standard?
- \*\* **Esercizio R8.8.** Da quali classi dipende la classe `Rectangle` della libreria standard?
- \* **Esercizio R8.9.** Catalogate come d'accesso o modificatori i metodi della classe `Scanner` usati in questo libro.
- \* **Esercizio R8.10.** Catalogate come d'accesso o modificatori i metodi della classe `Rectangle`.
- \* **Esercizio R8.11.** Quali delle seguenti classi sono immutabili?
- `Rectangle`
  - `String`
  - `Random`
- \* **Esercizio R8.12.** Quali delle seguenti classi sono immutabili?
- `PrintStream`
  - `Date`
  - `Integer`

- \*\* **Esercizio R8.13.** 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;
    }
}
```

- \*\*\* **Esercizio R8.14.** Idealmente, un metodo non dovrebbe avere effetti collaterali. Potete scrivere un programma in cui nessun metodo ha un effetto collaterale? Sarebbe un programma utile?

- \*\* **Esercizio R8.15.** Scrivete le pre-condizioni per i metodi seguenti, senza realizzarli.

- `public static double sqrt(double x)`
- `public static String romanNumeral(int n)`
- `public static double slope(Line2D.Double a)`
- `public static String weekday(int day)`

- \*\* **Esercizio R8.16.** Quali pre-condizioni hanno i seguenti metodi della libreria standard di Java?

- `Math.sqrt`
- `Math.tan`
- `Math.log`

- d. `Math.pow`
- e. `Math.abs`

**\*\* Esercizio R8.17.** Quali pre-condizioni hanno i seguenti metodi della libreria standard di Java?

- a. `Integer.parseInt(String s)`
- b. `StringTokenizer.nextToken()`
- c. `Random.nextInt(int n)`
- d. `String.substring(int m, int n)`

**\*\*\* Esercizio R8.18.** Dopo essere stato invocato con parametri che violano le proprie pre-condizioni, un metodo può restituire il controllo all'invocante oppure terminare l'esecuzione del programma (lanciando un'eccezione oppure un errore di asserzione). Fornite due esempi, tratti dalla libreria standard oppure da quanto visto in questo libro, di metodi che restituiscono un valore anche quando ricevono parametri non validi, e fornite altri due esempi di metodi che terminano l'esecuzione del programma.

**\*\* Esercizio R8.19.** Considerate una classe `CashRegister` con i metodi

- `public void enterPayment(int coinCount, Coin coinType)`
- `public double getTotalPayment()`

Fornite una post-condizione ragionevole per il metodo `enterPayment`. Di quali pre-condizioni avreste bisogno per fare in modo che la classe `CashRegister` possa garantire tale post-condizione?

**\*\* Esercizio R8.20.** Esamine il metodo seguente, creato per scambiare tra loro i valori di due numeri in virgola mobile:

```
public static void falseSwap(double a, double b)
{
    double temp = a;
    a = b;
    b = temp;
}

public static void main(String[] args)
{
    double x = 3;
    double y = 4;
    falseSwap(x, y);
    System.out.println(x + " " + y);
}
```

Perché il metodo non scambia il contenuto di `x` e di `y`?

**\*\*\* Esercizio R8.21.** In che modo potete scrivere un metodo che scambi il valore di due numeri in virgola mobile? Suggerimento: `Point2D.Double`.

**\*\* Esercizio R8.22.** 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;
}
```

- ★ **Esercizio R8.23.** 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`.

- ★ **Esercizio R8.24.** Provate a compilare il programma seguente e spiegate il messaggio d'errore che ottenete.

```
public class Print13
{
    public void print(int x)
    {
        System.out.println(x);
    }

    public static void main(String[] args)
    {
        int n = 13;
        print(n);
    }
}
```

- ★ **Esercizio R8.25.** Osservate i metodi della classe `Integer`. Quali sono statici? Perché?
- ★★ **Esercizio R8.26.** Osservate i metodi della classe `String` (ignorando quelli che ricevono un parametro di tipo `char[]`). Quali sono statici? Perché?
- ★★ **Esercizio 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 potreste migliorarla?
- ★★ **Esercizio R8.28.** Nella classe seguente, la variabile `n` ricorre in molti ambiti di visibilità. Quali dichiarazioni di `n` sono lecite e quali no?

```
public class X
{
    private int n;

    public int f()
    {
        int n = 1;
        return n;
    }

    public int g(int k)
```

```

{
    int a;
    for (int n = 1; n <= k; n++)
        a = a + n;
    return a;
}

public int h(int n)
{
    int b;
    for (int n = 1; n <= 10; n++)
        b = b + n;
    return b + n;
}

public int k(int n)
{
    if (n < 0)
    {
        int k = -n;
        int n = (int) (Math.sqrt(k));
        return n;
    }
    else return n;
}

public int m(int k)
{
    int a;
    for (int n = 1; n <= k; n++)
        a = a + n;
    for (int n = k; n >= 1; n++)
        a = a + n;
    return a;
}
}

```

- ★★ **Esercizio R8.29.** Qualsiasi programma Java può essere riscritto eliminando gli enunciati `import`. Spiegate come ciò sia possibile e riscrivete il file `RectangleComponent.java` del Capitolo 2 eliminando gli enunciati `import`.
- ★ **Esercizio R8.30.** Che cos'è il pacchetto predefinito? L'avete usato nella vostra programmazione prima di leggere questo capitolo?
- ★★ **Esercizio R8.31.** Come si comporta JUnit quando un metodo di collaudo lancia un'eccezione? Provate e descrivete ciò che scoprirete.

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

# 9

## Interfacce e polimorfismo

### Obiettivi del capitolo

- Saper dichiarare e usare interfacce
- Capire il concetto di polimorfismo
- 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, ma, per questo, si rendono spesso necessarie alcune modifiche allo schema di progetto dei componenti stessi. 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 e tale fenomeno prende il nome di *polimorfismo*.

## 9.1 Uso di interfacce per il riutilizzo del codice

Spesso è possibile rendere il codice più generale e maggiormente riutilizzabile focalizzando l'attenzione sulle operazioni essenziali che svolge: per esprimere queste operazioni comuni si utilizzano i *tipi interfaccia*.

Considerate la classe **DataSet** che abbiamo usato nel Capitolo 6 per calcolare il valore medio e il valore massimo di un insieme di valori di ingresso: purtroppo, poteva elaborare soltanto un insieme di *numeri*. Se volessimo esaminare conti bancari per trovare il conto con il saldo più elevato, dovremmo modificare la classe in questo modo:

```
public class DataSet // modificata per oggetti di tipo BankAccount
{
    private double sum;
    private BankAccount maximum;
    private int count;
    ...
    public void add(BankAccount x)
    {
        sum = sum + x.getBalance();
        if (count == 0 || maximum.getBalance() < x.getBalance())
            maximum = x;
        count++;
    }

    public BankAccount getMaximum()
    {
        return maximum;
    }
}
```

Immaginiamo, ora, di voler trovare, in un insieme di monete, quella avente il valore più elevato: dovremmo modificare nuovamente la classe **DataSet**.

```
public class DataSet // modificata per oggetti di tipo Coin
{
    private double sum;
    private Coin maximum;
    private int count;
    ...
    public void add(Coin x)
    {
        sum = sum + x.getValue();
        if (count == 0 || maximum.getValue() < x.getValue())
            maximum = x;
        count++;
    }

    public Coin getMaximum()
    {
        return maximum;
    }
}
```

Il meccanismo fondamentale per l'analisi dei dati è, evidentemente, lo stesso in tutti i casi, mentre cambiano i dettagli della misurazione che serve per il confronto: ci piacerebbe progettare un'unica classe che compia questa elaborazione su qualunque oggetto che possa essere misurato.

Supponete che le diverse classi potessero accordarsi sull'esistenza di un metodo `getMeasure` che fornisca la misura da usare nell'analisi dei dati, come il saldo per i conti bancari, il valore per le monete, e così via; in tal caso, potremmo realizzare un'unica classe `DataSet`, riutilizzabile, il cui metodo `add` assomiglierebbe a questo:

```
sum = sum + x.getMeasure();
if (count == 0 || maximum.getMeasure() < x.getMeasure())
    maximum = x;
count++;
```

Di che tipo è la variabile `x`? Idealmente, `x` dovrebbe potersi riferire a qualunque classe che abbia un metodo `getMeasure`.

Per esprimere il concetto di una funzionalità necessaria per una classe, in Java si usa un'*interfaccia*. Ecco la definizione di un'interfaccia che chiamiamo `Measurable`.

In Java, la dichiarazione di un'interfaccia contiene metodi, ma non il relativo codice.

```
public interface Measurable
{
    double getMeasure();
}
```

La dichiarazione di interfaccia elenca tutti i metodi richiesti dall'interfaccia stessa: questa richiede un solo metodo, ma, in generale, ne può richiedere più d'uno.

Notate che il tipo `Measurable` non fa parte della libreria standard: è un tipo di dati che è stato creato appositamente per questo libro, in modo da rendere maggiormente riutilizzabile la classe `DataSet`.

Un'interfaccia è simile a una classe, ma ci sono parecchie differenze importanti:

- Tutti i metodi di un'interfaccia sono *astratti*, cioè hanno un nome, un elenco di parametri, un tipo di valore restituito, ma non hanno un'implementazione.
- Tutti i metodi di un'interfaccia sono automaticamente pubblici.

## Sintassi di Java

### 9.1 Dichiarazione di interfaccia

#### Sintassi

```
public interface NomeInterfaccia
{
    firme dei metodi
}
```

#### Esempio

```
I metodi di un'interfaccia sono
automaticamente pubblici.      public interface Measurable
                                {
                                    double getMeasure();
                                }
```

Non viene fornita alcuna  
implementazione.

- Un'interfaccia non ha variabili di esemplare.

A questo punto possiamo usare il tipo `Measurable` per dichiarare le variabili `x` e `maximum`.

```
public class DataSet
{
    private double sum;
    private Measurable maximum;
    private int count;
    ...
    public void add(Measurable x)
    {
        sum = sum + x.getMeasure();
        if (count == 0 || maximum.getMeasure() < x.getMeasure())
            maximum = x;
        count++;
    }
    public Measurable getMaximum()
    {
        return maximum;
    }
}
```

*Per indicare che una classe realizza un'interfaccia si usa la parola riservata implements.*

Questa classe `DataSet` è utilizzabile per analizzare oggetti di qualsiasi classe che *realizzi* (o, come anche si dice, implementi) l'interfaccia `Measurable`. Una classe *realizza un'interfaccia* se la dichiara in una clausola `implements` e se, di conseguenza, realizza i metodi richiesti dall'interfaccia stessa.

```
public class BankAccount implements Measurable
{
    ...
    public double getMeasure()
    {
        return balance;
    }
}
```

Notate 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.

Analogamente, è semplice modificare la classe `Coin` perché realizzi l'interfaccia `Measurable`.

```
public class Coin implements Measurable
{
    public double getMeasure()
    {
        return value;
    }
    ...
}
```

In sostanza, l'interfaccia **Measurable** esprime ciò che hanno in comune tutti gli oggetti "misurabili" e tale astrazione rende più flessibile la classe **DataSet**, i cui esemplari possono essere utilizzati per analizzare collezioni di oggetti di *qualsiasi* classe che realizzzi l'interfaccia **Measurable**.

I tipi interfaccia vengono utilizzati per rendere il codice maggiormente riutilizzabile.

Si tratta di un utilizzo tipico delle interfacce: un fornitore di servizi (in questo caso, la classe **DataSet**) specifica un'interfaccia il cui rispetto consente di fruire del servizio e qualunque classe che sia conforme a tale interfaccia può essere servita, in modo assai simile a un frullatore, che garantisce la rotazione di qualsiasi attrezzo che si adegui all'interfaccia di connessione (Figura 1).

**Figura 1**

Gli attrezzi devono essere conformi all'interfaccia di connessione definita dal frullatore



## Sintassi di Java

## 9.2 Implementazione di interfaccia

### Sintassi

```
public class NomeClasse implements NomeInterfaccia1, NomeInterfaccia2, ...
{
    variabili di esemplare
    metodi
}
```

### Esempio

Variabili di esemplare  
di BankAccount

Altri metodi  
di BankAccount

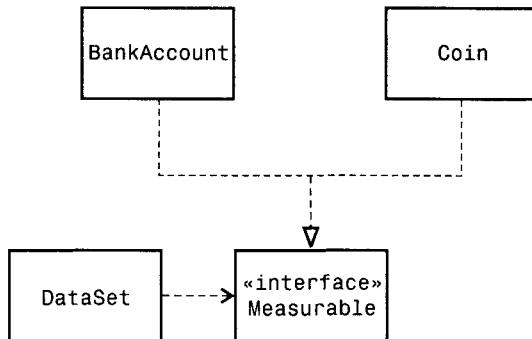
public class BankAccount implements Measurable {

```
    ...
    public double getMeasure()
    {
        return balance;
    }
}
```

Elenco di tutte le interfacce  
realizzate dalla classe.

Questo metodo fornisce l'implementazione  
del metodo dichiarato nell'interfaccia.

**Figura 2**  
 Schema UML della classe DataSet e delle classi che realizzano l'interfaccia Measurable



La Figura 2 mostra le relazioni che esistono tra la classe `DataSet`, l'interfaccia `Measurable` e le classi che la realizzano. Si noti che la classe `DataSet` dipende solamente dall'interfaccia `Measurable` e non è accoppiata alle classi `BankAccount` e `Coin`.

Nella notazione UML, le interfacce vengono contrassegnate dall'indicazione «`interface`», mentre una freccia tratteggiata con punta triangolare segnala la relazione, di tipo “è un”, che esiste tra una classe e un’interfaccia da essa realizzata. Occorre fare molta attenzione alla punta delle frecce: una linea tratteggiata con la freccia a V aperta indica, invece, una dipendenza (relazione “usa”).

#### File ch09/measure1/DataSetTester.java

```

/**
 * Questo programma collauda la classe DataSet.
 */
public class DataSetTester
{
    public static void main(String[] args)
    {
        DataSet bankData = new DataSet();

        bankData.add(new BankAccount(0));
        bankData.add(new BankAccount(10000));
        bankData.add(new BankAccount(2000));

        System.out.println("Average balance: " + bankData.getAverage());
        System.out.println("Expected: 4000");
        Measurable max = bankData.getMaximum();
        System.out.println("Highest balance: " + max.getMeasure());
        System.out.println("Expected: 10000");

        DataSet coinData = new DataSet();

        coinData.add(new Coin(0.25, "quarter"));
        coinData.add(new Coin(0.1, "dime"));
        coinData.add(new Coin(0.05, "nickel"));

        System.out.println("Average coin value: " + coinData.getAverage());
        System.out.println("Expected: 0.133");
        max = coinData.getMaximum();
        System.out.println("Highest coin value: " + max.getMeasure());
    }
}
  
```

```

        System.out.println("Expected: 0.25");
    }
}

```

## Esecuzione del programma

```

Average balance: 4000.0
Expected: 4000
Highest balance: 10000.0
Expected: 10000
Average coin value: 0.1333333333333333
Expected: 0.133
Highest coin value: 0.25
Expected: 0.25

```



## Auto-valutazione

- Immaginate di voler utilizzare la classe `DataSet` per trovare l'oggetto di tipo `Country` avente la popolazione più numerosa. Quale condizione deve essere soddisfatta dalla classe `Country`?
- Perché il metodo `add` della classe `DataSet` non può avere un parametro di tipo `Object`?



## Errori comuni 9.1

### Dimenticarsi di dichiarare pubblici i metodi realizzati

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 chiave `public` nella realizzazione di un metodo relativo a un'interfaccia è un errore comune:

```

public class BankAccount implements Measurable
{
    double getMeasure() // dovrebbe 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 pubblico il metodo.



## Argomenti avanzati 9.1

### Costanti nelle interfacce

Le interfacce non possono avere variabili di esemplare, ma al loro interno potete specificare *costanti*. Per esempio, l'interfaccia `SwingConstants` definisce diverse costanti, quali `SwingConstants.NORTH`, `SwingConstants.EAST` e simili.

Quando dichiarate una costante in un'interfaccia, potete (e dovreste) omettere le parole riservate `public static final`, perché tutte le variabili in un'interfaccia sono automaticamente `public static final`. Ad esempio:

```
public interface SwingConstants
{
    int NORTH = 1;
    int NORTH_EAST = 2;
    int EAST = 3;
    ...
}
```

## 9.2 Conversione di tipo fra classe e interfaccia

Le interfacce vengono utilizzate per esprimere le funzionalità comuni a più classi: in questo paragrafo vedremo come sia lecito effettuare conversioni di tipo fra una classe e un'interfaccia.

Guardate attentamente questa invocazione di metodo nel programma di collaudo del paragrafo precedente:

```
bankData.add(new BankAccount(10000));
```

Un oggetto di tipo `BankAccount` viene usato come parametro del metodo `add` della classe `DataSet`, ma quel metodo richiede un parametro di tipo `Measurable`:

```
public void add(Measurable x)
```

Potete effettuare conversioni dal tipo di una classe al tipo di un'interfaccia che sia realizzata dalla classe.

È lecita la conversione dal tipo `BankAccount` al tipo `Measurable`? In generale, è ammessa la conversione dal tipo di una classe al tipo di una delle interfacce realizzate dalla classe. Ad esempio:

```
BankAccount account = new BankAccount(10000);
Measurable meas = account; // va bene
```

Una variabile di tipo `Measurable` può anche riferirsi a un oggetto della classe `Coin` vista nel paragrafo precedente, dato che anch'essa realizza l'interfaccia `Measurable`.

```
Coin dime = new Coin(0.1, "dime");
Measurable meas = dime; // anche questo va bene
```

Però la classe `Rectangle` della libreria standard non realizza l'interfaccia `Measurable`, per cui questo enunciato di assegnazione è sbagliato:

```
Measurable meas = new Rectangle(5, 10, 20, 30); // ERRORE
```

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. Ciò accade, ad esempio, nel metodo `getMaximum` della classe `DataSet`, la quale memorizza l'oggetto avente la dimensione maggiore *come un riferimento di tipo Measurable*.

```
DataSet coinData = new DataSet();
coinData.add(new Coin(0.25, "quarter"));
coinData.add(new Coin(0.1, "dime"));
coinData.add(new Coin(0.05, "nickel"));
Measurable max = coinData.getMaximum();
```

Ora, cosa potete fare con il riferimento `max`? *Voi* sapete che si riferisce a un oggetto di tipo `Coin`, ma il compilatore non lo sa, per cui, ad esempio, non potete invocare il metodo `getName`:

```
String coinName = max.getName(); // ERRORE
```

Questa invocazione è un errore, perché il tipo `Measurable` non ha un metodo `getName`.

Tuttavia, dal momento che siete assolutamente certi che `max` si riferisca a un oggetto di tipo `Coin`, potete usare la notazione di *forzatura* (*cast*) per convertirlo al suo tipo originario:

```
Coin maxCoin = (Coin) max;
String name = maxCoin.getName();
```

Se vi siete sbagliati e l'oggetto in realtà non è una moneta, il vostro programma lancerà un'eccezione durante l'esecuzione e terminerà.

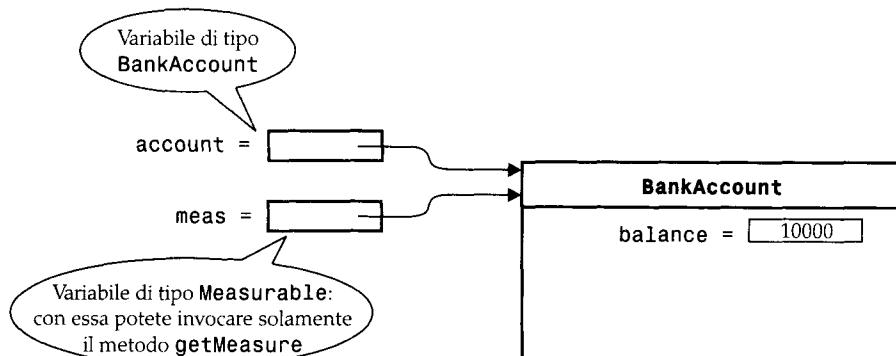
Questa notazione “*cast*” è la stessa che avete visto nel Capitolo 4 per effettuare conversioni fra tipi numerici diversi: per esempio, se `x` è un numero in virgola mobile, (`int`) `x` è la sua parte intera. In questo caso l'intento è analogo, ovvero fare la conversione da un tipo a un altro, però c'è una differenza sostanziale: quando convertite tipi numerici, ci può essere una *perdita di informazioni* e usate il cast per dire al compilatore che questo vi sta bene; d'altra parte, quando convertite riferimenti, *afrontate il rischio* di provocare il lancio di un'eccezione e dite al compilatore che siete disposti a correre tale rischio.

## Auto-valutazione

3. Potete usare un cast (`BankAccount`) `meas` per convertire una variabile `meas` di tipo `Measurable` in un riferimento di tipo `BankAccount`?
4. Se le classi `BankAccount` e `Coin` realizzano entrambe l'interfaccia `Measurable`, si può convertire un riferimento di tipo `Coin` in un riferimento di tipo `BankAccount`?

**Figura 3**

• Variabili di tipo riferimento a classe o riferimento a interfaccia





## Errori comuni 9.2

### Cercare di creare esemplari di un'interfaccia

Si possono definire variabili il cui tipo sia un'interfaccia, come in questo esempio:

```
Measurable meas;
```

ma non si possono *mai* costruire oggetti di tipo interfaccia:

```
Measurable meas = new Measurable(); // ERRORE
```

Le interfacce non sono classi e non esistono oggetti di tipo interfaccia. Se una variabile di tipo interfaccia fa riferimento a un oggetto, ciò significa che quell'oggetto deve appartenere a una classe che realizza quell'interfaccia:

```
Measurable meas = new BankAccount(); // va bene
```

## 9.3 Polimorfismo

In generale, quando più classi implementano la medesima interfaccia, ciascuna realizza i metodi propri dell'interfaccia in modi diversi. Come è possibile che, nel momento in cui viene invocato il metodo dell'interfaccia, venga eseguito il metodo corretto? In questo paragrafo risponderemo a questa domanda.

Vale la pena sottolineare ancora una volta che è assolutamente lecito, e in effetti molto comune, usare variabili il cui tipo sia un'interfaccia, come

```
Measurable meas;
```

Occorre, però, sempre ricordare che l'oggetto a cui si riferisce `meas` non è di tipo `Measurable`: infatti, *nessun oggetto* può essere di tipo `Measurable`. Il tipo dell'oggetto sarà sempre quello di una classe che realizza l'interfaccia `Measurable`, come `BankAccount`, `Coin` o qualche altra classe che abbia il metodo `getMeasure`.

```
meas = new BankAccount(10000); // va bene
meas = new Coin(0.1, "dime"); // va bene
```

Cosa potete fare con una variabile di tipo interfaccia, dato che non conoscete la classe dell'oggetto a cui si riferisce? Potete invocare i metodi dell'interfaccia:

```
double m = meas.getMeasure();
```

La classe `DataSet` sfrutta questa possibilità per calcolare la misura dell'oggetto aggiunto all'insieme, senza preoccuparsi di conoscere con precisione la natura dell'oggetto stesso. Cerchiamo ora di analizzare con maggiore attenzione l'invocazione del metodo `getMeasure`. *Quale* metodo `getMeasure`? Le classi `BankAccount` e `Coin` forniscono due *diverse* realizzazioni di quel metodo: in che modo viene invocato il metodo corretto, se chi invoca il metodo non sa con precisione a quale classe appartiene `meas`?

Quando la macchina virtuale invoca un metodo di esemplare, usa il metodo della classe che definisce il parametro implicito: si parla di ricerca dinamica del metodo.

Il polimorfismo è la capacità di gestire in modo omogeneo oggetti aventi comportamenti diversi.

La macchina virtuale Java identifica il metodo corretto guardando per prima cosa alla classe dell'oggetto effettivamente usato nell'invocazione, invocando il metodo avente quel nome e definito in quella classe. Se `meas` si riferisce a un oggetto di tipo `BankAccount`, allora viene invocato il metodo `BankAccount.getMeasure`; se, invece, `meas` si riferisce a un oggetto di tipo `Coin`, viene invocato il metodo `Coin.getMeasure`. Ciò significa che un'invocazione di metodo come la seguente:

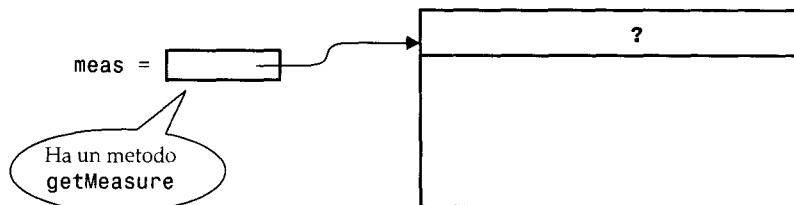
```
double m = meas.getMeasure();
```

può invocare metodi diversi in relazione al contenuto momentaneo di `meas`. Questo meccanismo di identificazione del metodo appropriato in un'invocazione è detto *ricerca dinamica del metodo* ("dynamic method lookup").

Questa ricerca dinamica del metodo rende possibile l'utilizzo di una tecnica di programmazione che prende il nome di *polimorfismo*, un termine che deriva dal greco e significa "multiforme": la stessa elaborazione funziona per oggetti di forme diverse e si adatta alla natura degli oggetti.

**Figura 4**

Un riferimento di tipo interfaccia può riferirsi a un oggetto di qualsiasi classe che realizza tale interfaccia.



## Auto-valutazione

5. Perché non si possono costruire oggetti di tipo `Measurable`?
6. Perché si può, invece, dichiarare una variabile di tipo `Measurable`?
7. Cosa visualizza questo frammento di codice? Perché costituisce un esempio di polimorfismo?

```
DataSet data = new DataSet();
data.add(new BankAccount(1000));
data.add(new Coin(0.1, "dime"));
System.out.println(data.getAverage());
```

## Esempi completi 9.1

### Analizzare sequenze di numeri

In questo esempio 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 o regole nascoste, oppure verificare se una sequenza è veramente casuale. Noi ci occuperemo di un problema molto semplice: la distribuzione dell'ultima cifra dei valori della sequenza. Per una sequenza assegnata, produrremo 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`.

```

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 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 (il programma completo si trova nella cartella ch09/sequence del pacchetto di file scaricabili per questo libro). 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
```

## 9.4 Usare interfacce di smistamento (*callback*)

In questo paragrafo introdurremo il concetto di “smistamento” o “richiamata” (*callback*), mostrando come possa rendere la classe `DataSet` 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 la classe `DataSet`, 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` realizzzi un'altra interfaccia: è una classe di libreria, che non potete modificare.
- Potete "misurare" un oggetto in un unico modo: se volete analizzare un insieme di conti bancari di risparmio sia in base al saldo che in base al tasso di interesse, siete bloccati.

Ripensiamo, quindi, alla classe `DataSet`: un insieme di dati deve poter misurare gli oggetti che vi vengono inseriti. Quando agli oggetti viene richiesto di essere di tipo `Measurable`, la responsabilità della misurazione ricade sugli oggetti stessi: da questo derivano le limitazioni che abbiamo notato.

Sarebbe meglio se potessimo fornire a un insieme di dati un metodo che misuri gli oggetti. In questo modo, elaborando rettangoli potremmo fornire un metodo che ne calcoli l'area, mentre elaborando conti bancari di risparmio potremmo fornire un metodo che ne ispezioni il tasso di interesse.

Il meccanismo di *callback* consente di specificare codice che verrà eseguito in un secondo momento.

Un metodo di questo tipo viene chiamato *callback* e costituisce un meccanismo per confezionare un frammento 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 "minimo comun denominatore" di tutte le classi in Java. Discuteremo il tipo `Object` con maggiore dettaglio nel Capitolo 10.

Il codice che effettua l'invocazione del metodo di smistamento riceve un oggetto di una classe che implementa questa interfaccia. Nel nostro caso, si costruisce un esemplare della classe `DataSet` migliorata fornendo un oggetto di tipo `Measurer` (cioè un oggetto di una classe che realizza l'interfaccia `Measurer`), che viene memorizzato nella variabile di esemplare `measurer`:

```
public DataSet(Measurer aMeasurer)
{
    sum = 0;
    count = 0;
    maximum = null;
    measurer = aMeasurer;
}
```

La variabile `measurer` viene, poi, utilizzata per eseguire effettivamente le misurazioni, in questo modo:

```
public void add(Object x)
{
    sum = sum + measurer.measure(x);
    if (count == 0 || measurer.measure(maximum) < measurer.measure(x))
        maximum = x;
    count++;
}
```

La classe `DataSet` richiama semplicemente il metodo `measure` ogni volta che deve misurare un oggetto qualsiasi.

Infine, si ottiene un misuratore specifico realizzando l'interfaccia `Measurer`. Ad esempio, ecco come classificare rettangoli misurandone l'area. Definite la classe:

```
public class RectangleMeasurer implements Measurer
{
    public double measure(Object anObject)
    {
        Rectangle aRectangle = (Rectangle) anObject;
        double area = aRectangle.getWidth() * aRectangle.getHeight();
        return area;
    }
}
```

Notate che il metodo `measure` deve accettare un 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 `RectangleMeasurer`? Vi serve per costruire un esemplare di `DataSet` che confronti rettangoli in base alla loro area. Costruite un oggetto di tipo `RectangleMeasurer` e passatelo al costruttore di `DataSet`:

```
Measurer m = new RectangleMeasurer();
DataSet data = new DataSet(m);
```

Successivamente, aggiungete rettangoli all'insieme di dati.

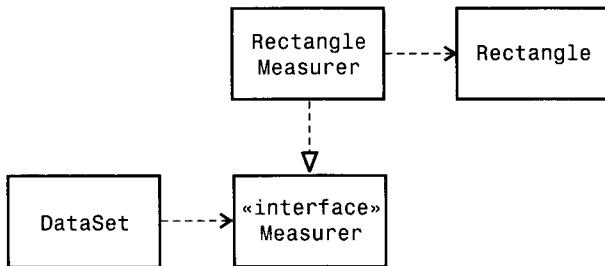
```
data.add(new Rectangle(5, 10, 20, 30));
data.add(new Rectangle(10, 20, 30, 40));
...
```

L'insieme di dati chiederà all'oggetto di tipo `RectangleMeasurer` di misurare i rettangoli: in altre parole, l'insieme di dati "smista" le misurazioni all'oggetto `RectangleMeasurer`.

La Figura 5 mostra il diagramma UML delle classi e delle interfacce usate in questa soluzione. Come nella Figura 2, la classe `DataSet` è disaccoppiata dalla classe `Rectangle`, di cui elabora oggetti. Tuttavia, diversamente da quanto accadeva in Figura 2, la classe `Rectangle` non è più accoppiata a un'altra classe: per elaborare rettangoli, fornite una

**Figura 5**

Schema UML della classe DataSet e dell'interfaccia Measurer



piccola classe “ausiliaria”, `RectangleMeasurer`, che ha solamente lo scopo di aiutare la classe `DataSet` a misurare i propri oggetti.

#### File ch09/measure2/Measurer.java

```

/**
 * Descrive qualsiasi classe i cui esemplari possano
 * misurare altri oggetti.
 */
public interface Measurer
{
    /**
     * Calcola la misura di un oggetto.
     * @param anObject l'oggetto da misurare
     * @return la misura
     */
    double measure(Object anObject);
}
  
```

#### File ch09/measure2/RectangleMeasurer.java

```

import java.awt.Rectangle;

/**
 * Gli oggetti di questa classe misurano rettangoli in base alla loro area.
 */
public class RectangleMeasurer implements Measurer
{
    public double measure(Object anObject)
    {
        Rectangle aRectangle = (Rectangle) anObject;
        double area = aRectangle.getWidth() * aRectangle.getHeight();
        return area;
    }
}
  
```

#### File ch09/measure2/DataSet.java

```

/**
 * Calcola la media di un insieme di valori.
 */
public class DataSet
{
}
  
```

```

private double sum;
private Object maximum;
private int count;
private Measurer measurer;

/**
 * Costruisce un insieme vuoto di dati con un misuratore assegnato.
 * @param aMeasurer il misuratore che viene usato
 *                  per misurare i valori dei dati
 */
public DataSet(Measurer aMeasurer)
{
    sum = 0;
    count = 0;
    maximum = null;
    measurer = aMeasurer;
}

/**
 * Aggiunge un valore all'insieme di dati.
 * @param x il valore da aggiungere
 */
public void add(Object x)
{
    sum = sum + measurer.measure(x);
    if (count == 0 || measurer.measure(maximum) < measurer.measure(x))
        maximum = x;
    count++;
}

/**
 * Restituisce la media dei dati inseriti.
 * @return la media dei valori (0 se non ce ne sono)
 */
public double getAverage()
{
    if (count == 0) return 0;
    else return sum / count;
}

/**
 * Restituisce il valore maggiore tra quelli inseriti.
 * @return il valore maggiore, oppure null se non ci sono valori
 */
public Object getMaximum()
{
    return maximum;
}
}

```

### File ch09/measure2/DataSetTester2.java

```

import java.awt.Rectangle;

/**
 * Questo programma illustra l'utilizzo di un oggetto di tipo Measurer.

```

```

*/
public class DataSetTester2
{
    public static void main(String[] args)
    {
        Measurer m = new RectangleMeasurer();

        DataSet data = new DataSet(m);

        data.add(new Rectangle(5, 10, 20, 30));
        data.add(new Rectangle(10, 20, 30, 40));
        data.add(new Rectangle(20, 30, 5, 15));

        System.out.println("Average area: " + data.getAverage());
        System.out.println("Expected: 625");

        Rectangle max = (Rectangle) data.getMaximum();
        System.out.println("Maximum area rectangle: " + max);
        System.out.println("Expected: "
            + "java.awt.Rectangle[x=10,y=20,width=30,height=40]");
    }
}

```

### Esecuzione del programma

```

Average area: 625
Expected: 625
Maximum area rectangle: java.awt.Rectangle[x=10,y=20,width=30,height=40]
Expected: java.awt.Rectangle[x=10,y=20,width=30,height=40]

```



### Auto-valutazione

8. Immaginate di voler utilizzare la classe `DataSet` del Paragrafo 9.1 per trovare la stringa di lunghezza maggiore in un insieme di dati forniti in ingresso. Perché ciò non è possibile?
9. Come potete utilizzare la classe `DataSet` di questo paragrafo per trovare la stringa di lunghezza maggiore in un insieme di dati forniti in ingresso?
10. Perché il metodo `measurer` dell'interfaccia `Measurer` ha un parametro in più del metodo `getMeasure` dell'interfaccia `Measurable`?

## 9.5 Classi interne

La classe `RectangleMeasurer` è veramente banale: ne abbiamo bisogno soltanto perché `DataSet` richiede un oggetto di una classe che realizzi 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 DataSetTester3
{
    public static void main(String[] args)
    {

```

```

        class RectangleMeasurer implements Measurer
        {
        ...
        }

        Measurer m = new RectangleMeasurer();
        DataSet data = new DataSet(m);
        ...
    }
}

```

Una classe interna viene dichiarata dentro un'altra classe.

Le classi interne sono molto utilizzate per classi con scopo molto limitato, che non hanno bisogno di essere visibili in altre porzioni del programma.

Una classe dichiarata all'interno di un'altra classe, come la classe `RectangleMeasurer` di questo esempio, viene detta *classe interna (inner class)*. Questa disposizione segnala al lettore del vostro programma che la classe `RectangleMeasurer` non ha interesse al di fuori dell'ambito 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 DataSetTester3
{
    class RectangleMeasurer implements Measurer
    {
    ...

    public static void main(String[] args)
    {
        Measurer m = new RectangleMeasurer();
        DataSet data = new DataSet(m);
        ...
    }
}

```

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 `DataSetTester3$1$RectangleMeasurer.class`. I nomi esatti non sono importanti: ciò che importa è che il compilatore traduce una classe interna in un normale file di classe.

### File ch09/measure3/DataSetTester3.java

```

import java.awt.Rectangle;

/**
 * Questo programma illustra l'utilizzo di una classe interna.
 */
public class DataSetTester3
{
    public static void main(String[] args)
    {
        class RectangleMeasurer implements Measurer

```

```

    {
        public double measure(Object anObject)
        {
            Rectangle aRectangle = (Rectangle) anObject;
            double area = aRectangle.getWidth()
                         * aRectangle.getHeight();
            return area;
        }
    }

    Measurer m = new RectangleMeasurer();

    DataSet data = new DataSet(m);

    data.add(new Rectangle(5, 10, 20, 30));
    data.add(new Rectangle(10, 20, 30, 40));
    data.add(new Rectangle(20, 30, 5, 15));

    System.out.println("Average area: " + data.getAverage());
    System.out.println("Expected: 625");

    Rectangle max = (Rectangle) data.getMaximum();
    System.out.println("Maximum area rectangle: " + max);
    System.out.println("Expected: "
                       + "java.awt.Rectangle[x=10,y=20,width=30,height=40]");
}
}
}

```



## Auto-valutazione

11. Perché si usa una classe interna invece di una classe normale?
12. Quanti file di classe vengono generati compilando il programma `DataSetTester3`?



## Argomenti avanzati 9.2

### Classi interne 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 moneta non viene più utilizzata all'interno dello stesso metodo, potete sostituire questi enunciati:

```
Coin aCoin = new Coin(0.1, "dime");
data.add(aCoin);
```

con il seguente:

```
data.add(new Coin(0.1, "dime"));
```

L'oggetto `new Coin(0.1, "dime")` è un *oggetto anonimo*. Ai programmati piacciono gli oggetti anonimi, perché non devono sforzarsi di trovare loro un nome: se avete mai affrontato una decisione sofferta per chiamare una moneta `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 della classe `RectangleMeasurer`, essa 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)
{
    // costruisci 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();
        }
    };
    DataSet data = new DataSet(m);
    ...
}
```

Questo codice significa: costruisci un oggetto di una classe che realizza l'interfaccia `Measurer`, definendo il metodo `measure` come indicato. Alcuni programmati prediligono questo stile, ma non lo useremo in questo libro.

## 9.6 Oggetti semplificati

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 per gestire un registro scolastico, con i risultati di varie prove di più studenti. Per questo, ci serve una 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` come prefisso per il nome di tale interfaccia.

```
public interface IGradeBook
{
    void addScore(int studentId, double score);
```

**Un oggetto mock fornisce gli stessi servizi di un altro oggetto, in modo semplificato.**



## Note di cronaca 9.1

### Sistemi operativi

Senza un sistema operativo, un computer sarebbe inutile: come minimo, avete bisogno di un sistema operativo per individuare i file e per iniziare l'esecuzione di programmi. I programmi che eseguite necessitano di servizi da parte del sistema operativo, per accedere ai dispositivi e per interagire con altri programmi; i sistemi operativi di grandi computer devono fornire ancora più servizi di quelli che operano su personal computer.

Ecco alcuni servizi tipici:

- *Caricamento dei programmi.* Tutti i sistemi operativi forniscono modalità per mettere in esecuzione (“lanciare”) programmi applicativi. L’utente indica quale programma bisogna eseguire, generalmente scrivendo sulla tastiera il nome del programma oppure selezionando un’icona, dopodiché il sistema operativo individua il codice del programma, lo carica in memoria e lo avvia.
- *Gestione di file.* Dal punto di vista elettronico, un dispositivo di memorizzazione, quale un disco rigido, è semplicemente un apparecchio in grado di immagazzinare un’enorme quantità di zeri e di uni. È compito del sistema operativo applicare qualche struttura alla disposizione dei dati memorizzati, per organizzarli in file, cartelle, e così via. Inoltre, il sistema operativo deve aggiungere sicurezza e ridondanza al sistema di file e cartelle, affinché un’interruzione di energia elettrica non metta a repentaglio tutto il contenuto di un disco rigido: sotto questo aspetto, alcuni sistemi operativi sono più efficaci di altri.
- *Memoria virtuale.* La RAM è costosa e pochi computer ne hanno a sufficienza per contenere tutti quei programmi, con i loro dati, che un utente vuole tipicamente eseguire contemporaneamente. La maggior parte dei sistemi operativi aumenta la memoria disponibile depositando alcuni dati nel disco rigido, benché i programmi applicativi non si rendano conto di quanto succede. Quando un programma accede a porzioni di dati che non sono attualmente nella RAM, il processore lo rileva e lo comunica al sistema operativo, che richiama i dati richiesti dal disco rigido alla RAM, togliendo contemporaneamente dalla memoria un blocco di pari dimensioni, che non abbia avuto accessi per qualche tempo.
- *Gestione di più utenti.* Il sistema operativo di computer grandi e potenti consente l’accesso simultaneo da parte di più utenti. Ciascun utente è connesso al computer attraverso un terminale distinto: il sistema operativo ne accerta l’identità, verificando che abbia un account valido e che fornisca la prevista parola d’accesso (*password*), quindi fornisce una piccola *quota* di tempo di elaborazione del processore, per poi servire l’utente successivo.
- *Gestione di più processi (multitasking).* Anche nel caso in cui siate l’unico utente di un computer, potreste volere eseguire più applicazioni, per esempio leggere la posta elettronica in una finestra ed eseguire il compilatore Java in un’altra. Il sistema operativo ha la responsabilità di suddividere il tempo di elaborazione del processore fra le applicazioni in esecuzione, in modo che ciascuna possa procedere.
- *Stampa.* Il sistema operativo accoda le richieste di stampa che vengono inviate dalle diverse applicazioni. È un’operazione necessaria per garantire che le pagine stampate non contengano un miscuglio di parole inviate contemporaneamente da programmi distinti.
- *Finestre.* Molti sistemi operativi si presentano all’utente tramite un *desktop* (“scrivania”), costituito da più finestre. Il sistema operativo gestisce la posizione e l’aspetto delle finestre, mentre le applicazioni sono responsabili di quanto contenuto all’interno.
- *Font.* Per rappresentare il testo sullo schermo e nelle stampe, bisogna definire la forma dei caratteri, specialmente nel caso di programmi che possono visualizzare più stili tipografici e più dimensioni di carattere. I sistemi operativi moderni contengono una raccolta di font centralizzata.
- *Comunicazione fra programmi.* Il sistema operativo può agevolare il trasferimento di informazioni fra programmi. Questo passaggio può svolgersi mediante il *taglia e*

*incolla* o tramite la *comunicazione fra processi*. Il taglia e incolla è un trasferimento di dati iniziato dall'utente, che copia dati da un'applicazione a una zona di transito (spesso chiamata *clipboard*, “lavagna per appunti”), gestita dal sistema operativo,

- quindi ne inserisce il contenuto in un'altra applicazione. La comunicazione fra processi viene avviata da applicazioni che trasferiscono dati senza il coinvolgimento diretto dell'utente.
- *Connessioni di rete*. Il sistema operativo fornisce protocolli e ser-

vizi per abilitare le applicazioni a ottenere informazioni da altri computer connessi alla rete.

Attualmente, i sistemi operativi più diffusi sono Linux, Macintosh OS e Microsoft Windows.

```
    double getAverageScore(int studentId);
    void save(String filename);
    ...
}
```

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 (“mock”), 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 il parametro 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
    }
    ...
}
```

La classe *mock* e la corrispondente classe completa realizzano la medesima interfaccia.

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

13. Perché è necessario che la classe semplificata e la classe effettiva realizzino la medesima interfaccia?
14. Perché l'utilizzo degli oggetti `mock` è particolarmente efficace quando le classi `GradeBook` e `GradingProgram` sono sviluppate da due programmatore diversi?

## 9.7 Eventi: ricevitori e sorgenti

In questo paragrafo e nei successivi torniamo sulla programmazione grafica: vedrete come usare le interfacce 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. Tuttavia, i programmi che usate quotidianamente non funzionano in questo modo: in un'applicazione dotata di una moderna 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 programmi Java capaci di rispondere a eventi generati dall'utente attraverso l'interfaccia grafica del programma, come selezioni di voci in menu e pressioni di pulsanti del mouse. La raccolta di strumenti Java per la gestione delle finestre (*Java windowing toolkit*) ha un meccanismo molto articolato, che permette a un programma di specificare gli eventi che gli interessano, oltre a indicare quali oggetti devono essere avvertiti quando si verifica uno di tali 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 può generare un numero enorme di eventi: per esempio, ogni volta che il mouse si sposta di un intervallo minimo all'interno di una finestra, viene generato un evento di tipo “movimento del mouse”, mentre ad 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

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 appartiene a una classe progettata dal programmatore dell'applicazione e i suoi metodi descrivono le azioni da compiere in risposta a un evento.

grafiche su un canovaccio virtuale, sarà necessario tenere traccia con grande attenzione di tutti i singoli eventi del mouse.

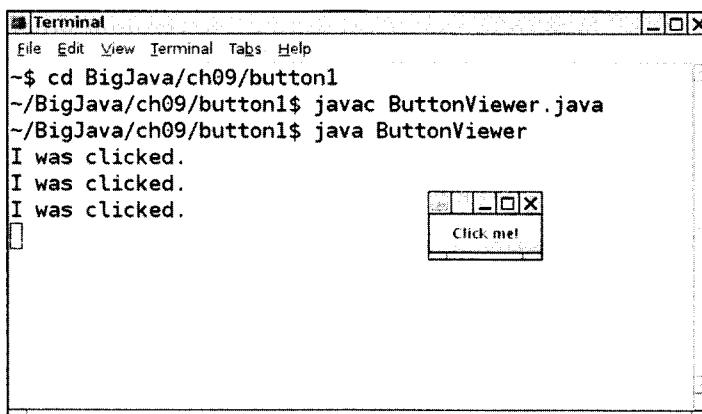
Ogni programma deve indicare, mediante l'installazione di oggetti *ricevitori di eventi* ("event listener"), quali eventi gradisce ricevere. Ciascun oggetto che funge da ricevitore appartiene a una classe progettata da voi e i suoi metodi contengono le istruzioni che vanno eseguite quando accade quel particolare evento che si intende ricevere.

Per installare un ricevitore, dovete conoscere la *sorgente dell'evento*, 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 sembra piuttosto 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 6.

Le sorgenti di eventi generano  
segnalazioni relative agli eventi.  
Quando ne avviene uno,  
lo segnalano a tutti i ricevitori  
interessati.

**Figura 6**  
Realizzazione  
di un ricevitore di azioni



I ricevitori interessati agli eventi di un pulsante grafico devono appartenere a una classe che realizza 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. Ecco un esempio molto semplice di una classe di questo tipo:

### File ch09/button1/ClickListener.java

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
```

Usate componenti di tipo JButton  
per realizzare pulsanti grafici  
e connettete  
un ActionListener  
a ogni pulsante.

```

        Un ricevitore di azioni che visualizza un messaggio.
*/
public class ClickListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        System.out.println("I was clicked");
    }
}

```

Ignoriamo il 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:

```

ActionListener listener = new ClickListener();
button.addActionListener(listener);

```

Ogni volta che il pulsante viene premuto, 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 la classe `DataSet` invoca il metodo `measure` ogni volta che ha bisogno di misurare un oggetto.

La classe `ButtonViewer`, di cui qui trovate il codice, costruisce un frame con un pulsante, al quale aggiunge un oggetto di tipo `ClickListener`. Potete collaudare questo programma aprendo una finestra di console, eseguendovi `ButtonViewer`, premendo il pulsante grafico con il mouse e osservando i messaggi visualizzati, come si può vedere nella Figura 6.

### File ch09/button1/ButtonViewer.java

```

import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

/**
 * Questo programma mostra come si installa un ricevitore di azioni.
 */
public class ButtonViewer
{
    private static final int FRAME_WIDTH = 100;
    private static final int FRAME_HEIGHT = 60;

    public static void main(String[] args)
    {
        JFrame frame = new JFrame();
        JButton button = new JButton("Click me!");
        frame.add(button);
    }
}

```

```

        ActionListener listener = new ClickListener();
        button.addActionListener(listener);

        frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

```



## Auto-valutazione

15. Nel programma `ButtonViewer`, quali sono gli oggetti che rappresentano la sorgente di eventi e il ricevitore di eventi?
16. Perché è lecito assegnare un oggetto di tipo `ClickListener` a una variabile di tipo `ActionListener`?



## Errori comuni 9.3

### Realizzare un metodo modificandone il tipo dei parametri

Quando realizzate un'interfaccia, dovete dichiarare ciascun metodo *esattamente* come specificato nell'interfaccia: 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, avete dimenticato il parametro ActionEvent
    {
        ...
    }
}

```

Per quanto riguarda il compilatore, questa classe è manchevole della definizione del metodo

```
public void actionPerformed(ActionEvent event)
```

Dovete leggere con cura il messaggio d'errore, facendo attenzione ai tipi dei parametri e del valore restituito: scoprirete così il vostro errore.

## 9.8 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 le motivazioni principali per questa strategia. Innanzitutto, banalmente, la classe che riceve gli eventi 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 un blocco annidato.

Ricordate che un blocco è un gruppo di enunciati racchiusi tra parentesi graffe. Se un blocco è annidato all'interno di un altro blocco, il blocco più interno ha accesso a tutte le variabili del blocco circostante:

```
{ // blocco circostante
    BankAccount account = new BankAccount();
    if (...)

        { // blocco interno
            ...
            // l'accesso alla variabile del blocco circostante è consentito
            account.deposit(interest);
            ...
        } // fine del blocco interno
        ...
    } // fine del blocco circostante
```

I metodi di una classe interna  
possono accedere alle variabili locali  
e di esemplare dell'ambito  
di visibilità circostante.

Lo stesso tipo di annidamento funziona per le classi interne: tranne per alcune restrizioni di tipo tecnico, che vedremo più avanti in questo stesso paragrafo, i metodi di una classe interna possono accedere alle variabili dell'ambito di visibilità circostante, una caratteristica molto utile per la 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 una classe interna devono essere dichiarate final.

C'è però un espediente tecnico da conoscere: una classe interna può accedere a variabili *locali* dell'ambito di visibilità circostante solo se sono state dichiarate *final*. Ciò sembra una restrizione, ma in pratica solitamente non lo è. Ricordate che una variabile di tipo riferimento a oggetto è *final* quando si riferisce sempre al medesimo oggetto: lo stato dell'oggetto può cambiare, ma la variabile non può riferirsi a un oggetto diverso. Ad esempio, nel nostro programma non abbiamo mai avuto l'intenzione di fare in modo che la variabile *account* si riferisse a conti bancari diversi con il trascorrere del tempo, per cui non c'è alcun problema nel dichiararla *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 ch09/button2/InvestmentViewer1.java

```

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

/**
 * Questo programma illustra il funzionamento di una classe
 * interna che accede a una variabile di un blocco circostante.
 */
public class InvestmentViewer1
{
    private static final int FRAME_WIDTH = 120;
    private static final int FRAME_HEIGHT = 60;

    private static final double INTEREST_RATE = 10;
    private static final double INITIAL_BALANCE = 1000;

    public static void main(String[] args)
    {
        JFrame frame = new JFrame();

        // il pulsante che fa partire il calcolo
        JButton button = new JButton("Add Interest");
        frame.add(button);

        // l'applicazione accredita gli interessi a questo conto bancario
        final bankAccount account = new BankAccount(INITIAL_BALANCE);

        class AddInterestListener implements ActionListener
        {

```

```

public void actionPerformed(ActionEvent event)
{
    // il metodo del ricevitore accede alla variabile
    // account del blocco circostante
    double interest = account.getBalance() * INTEREST_RATE / 100;
    account.deposit(interest);
    System.out.println("balance: " + account.getBalance());
}
};

ActionListener listener = new AddInterestListener();
button.addActionListener(listener);

frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
}
}

```

### Esecuzione del programma

```

balance: 1100.0
balance: 1210.0
balance: 1331.0
balance: 1464.1

```



### Auto-valutazione

17. 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?
18. Se una classe interna accede a una variabile locale che si trova in un ambito di visibilità circostante, quale speciale regola occorre applicare?

## 9.9 Costruire applicazioni dotate di pulsanti



In questo paragrafo vedrete come si struttura 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 gli interessi maturati sul conto bancario e viene visualizzato il saldo aggiornato, come si può vedere nella Figura 7.

Per prima cosa costruiamo un oggetto di tipo `JButton`, fornendo al costruttore l'etichetta del pulsante:

```
JButton button = new JButton("Add Interest");
```

**Figura 7**

Un'applicazione dotata  
di pulsante grafico



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 (di tipo *frame*) 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  
del mouse vanno specificate  
mediante classi  
che realizzano 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 realizzzi l'interfaccia *ActionListener*, inserendo all'interno del metodo *actionPerformed* di tale classe l'azione desiderata. La nostra classe di gestione degli eventi aggiunge gli interessi al 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* elabora 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*. Dobbiamo, quindi, 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 realizzano ricevitori di eventi sotto forma di classi interne.

### File ch09/button3/InvestmentViewer2.java

```

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

/**
 * Questo programma visualizza la crescita di un investimento.
 */
public class InvestmentViewer2
{
    private static final int FRAME_WIDTH = 400;
    private static final int FRAME_HEIGHT = 100;

    private static final double INTEREST_RATE = 10;
    private static final double INITIAL_BALANCE = 1000;

    public static void main(String[] args)
    {
        JFrame frame = new JFrame();

        // il pulsante che fa partire l'elaborazione
        JButton button = new JButton("Add Interest");

        // l'applicazione aggiunge gli interessi a questo conto bancario
        final BankAccount account = new BankAccount(INITIAL_BALANCE);

        // l'etichetta che visualizza i risultati
        final JLabel label = new JLabel("balance = " + account.getBalance());

        // il pannello che contiene i componenti dell'interfaccia grafica

```

```

JPanel panel = new JPanel();
panel.add(button);
panel.add(label);
frame.add(panel);

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);

frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
}
}

```

## Auto-valutazione

19. Come si fa a visualizzare il messaggio "balance = ..." alla sinistra del pulsante "Add Interest"?
20. Perché non è necessario dichiarare `final` anche la variabile `button`?

## Errori comuni 9.4

### Dimenticare di associare un ricevitore

Se, eseguendo il vostro programma, scoprirete che i pulsanti sembrano non funzionare, controllate bene di avere 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.

## Consigli per la produttività 9.1

### 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 dovete 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 fanno a meno delle 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 grossi problemi. Prima di tutto, separa la dichiarazione dei pulsanti dalle azioni relative ai pulsanti stessi. 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.

Errori comuni 9.5

**Per impostazione predefinita, i componenti hanno dimensioni nulle**

Quando aggiungete a un pannello un componente disegnato, come quello che visualizza un'automobile, dovete fare attenzione, perché la procedura è uguale a quella vista per aggiungere un pulsante o un'etichetta.

```
panel.add(button);
panel.add(label);
panel.add(carComponent);
```

ma la dimensione predefinita di un componente è pari a 0 per 0 pixel, per cui il componente che disegna l'automobile non sarebbe visibile. La soluzione consiste nell'invocare il metodo `setPreferredSize`, in questo modo:

## 9.10 Elaborare 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, può darsi che, in un'animazione, vogliate aggiornare una scena dieci volte al secondo, ridisegnando l'immagine per dare l'illusione del movimento.

Un temporizzatore genera eventi a intervalli di tempo fissi.

Quando usate un temporizzatore, dovete fornire al suo costruttore la frequenza degli eventi e un oggetto di una classe che realizzi l'interfaccia `ActionListener` e che contenga l'azione desiderata all'interno del proprio metodo `actionPerformed`. Infine, fate partire il temporizzatore.

```
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 `moveBy` sposta il rettangolo della quantità specificata.

### File ch09/timer/RectangleComponent.java

```
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import javax.swing.JComponent;

/**
 * Questo componente visualizza un rettangolo
 * che può essere spostato.
 */
public class RectangleComponent extends JComponent
{
    private static final int BOX_X = 100;
    private static final int BOX_Y = 100;
    private static final int BOX_WIDTH = 20;
    private static final int BOX_HEIGHT = 30;
```

```

private Rectangle box;

public RectangleComponent()
{
    // il rettangolo che viene disegnato dal metodo paint
    box = new Rectangle(BOX_X, BOX_Y, BOX_WIDTH, BOX_HEIGHT);
}

public void paintComponent(Graphics g)
{
    Graphics2D g2 = (Graphics2D) g;

    g2.draw(box);
}

/**
 * Sposta il rettangolo della quantità specificata.
 * @param x l'entità dello spostamento nella direzione x
 * @param y l'entità dello spostamento nella direzione y
 */
public void moveBy(int dx, int dy)
{
    box.translate(dx, dy);
    repaint();
}
}

```

Il metodo `repaint` chiede a un componente di ridisegnare se stesso: invocato ogni volta che modificate le forme grafiche disegnate dal metodo `paintComponent`.

Note l'invocazione di `repaint` nel metodo `moveBy`: 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 i valori di quattro coordinate. 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.moveBy(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 ch09/timer/RectangleMover.java

```

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.Timer;

/**
 * Questo programma sposta un rettangolo.
 */
public class RectangleMover
{
    private static final int FRAME_WIDTH = 300;
    private static final int FRAME_HEIGHT = 400;

```

```

public static void main(String[] args)
{
    JFrame frame = new JFrame();

    frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
    frame.setTitle("An animated rectangle");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    final RectangleComponent component = new RectangleComponent();
    frame.add(component);

    frame.setVisible(true);

    class TimerListener implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            component.moveBy(1, 1);
        }
    }

    ActionListener listener = new TimerListener();

    final int DELAY = 100; // millisecondi tra due eventi
    Timer t = new Timer(DELAY, listener);
    t.start();
}
}

```

## Auto-valutazione

21. Perché un temporizzatore ha bisogno di un oggetto ricevitore di eventi?
22. Cosa succederebbe se nel metodo `moveBy` mancasse l'invocazione del metodo `repaint`?

## Errori comuni 9.6

### 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 dovrete 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.

## 9.11 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 comunemente 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
{
    // realizzazione dei cinque metodi
}

MouseListener listener = new MyMouseListener()
component.addMouseListener(listener);
```

Nel nostro programma di esempio, ogni volta che l'utente preme e rilascia il pulsante del mouse (cioè fa “click”) sul componente rettangolare, vogliamo che il rettangolo si sposti e si posizionи nel punto in cui si trova il mouse. Per prima cosa modifichiamo la classe `RectangleComponent`, aggiungendole un metodo `moveTo` che sposti il rettangolo in una nuova posizione.

### File ch09/mouse/RectangleComponent.java

```

import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import javax.swing.JComponent;

/**
 * Questo componente visualizza un rettangolo che può essere spostato.
 */
public class RectangleComponent extends JComponent
{
    private static final int BOX_X = 100;
    private static final int BOX_Y = 100;
    private static final int BOX_WIDTH = 20;
    private static final int BOX_HEIGHT = 30;

    private Rectangle box;

    public RectangleComponent()
    {
        // il rettangolo disegnato dal metodo paint
        box = new Rectangle(BOX_X, BOX_Y, BOX_WIDTH, BOX_HEIGHT);
    }

    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;

        g2.draw(box);
    }

    /**
     * Sposta il rettangolo nella posizione indicata.
     * @param x la coordinata x della nuova posizione
     * @param y la coordinata y della nuova posizione
     */
    public void moveTo(int x, int y)
    {
        box.setLocation(x, y);
        repaint();
    }
}

```

Notate l'invocazione di `repaint` nel metodo `moveTo`, che, 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.moveTo(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 `RectangleComponentViewer`: 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 8.

#### File ch09/mouse/RectangleComponentViewer.java

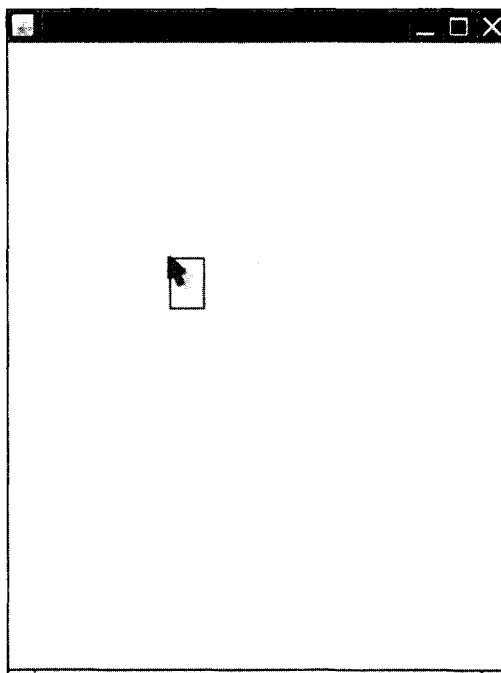
```

import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;

```

**Figura 8**

Un “click” del mouse sposta  
il rettangolo



```


    /**
     * Questo programma visualizza un RectangleComponent.
    */
public class RectangleComponentViewer
{
    private static final int FRAME_WIDTH = 300;
    private static final int FRAME_HEIGHT = 400;

    public static void main(String[] args)
    {
        final RectangleComponent component = new RectangleComponent();

        // associa il ricevitore di eventi di pressione di un pulsante del mouse

        class MousePressListener implements MouseListener
        {
            public void mousePressed(MouseEvent event)
            {
                int x = event.getX();
                int y = event.getY();
                component.moveTo(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) {}
        }

        MouseListener listener = new MousePressListener();
        component.addMouseListener(listener);

        JFrame frame = new JFrame();
        frame.add(component);

        frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}


```

## Auto-valutazione

23. Perché nella classe RectangleComponent il metodo `moveBy` è stato sostituito dal metodo `moveTo`?
24. Perché la classe `MousePressListener` deve avere cinque metodi?

## Argomenti avanzati 9.3

### 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



## Note di cronaca 9.2

### Linguaggi di programmazione

Attualmente esistono molte centinaia di linguaggi di programmazione e, in effetti, ciò è piuttosto sorprendente. Un linguaggio di alto livello ha lo scopo di fornire uno strumento di programmazione che sia indipendente dall'insieme delle istruzioni di un determinato processore, in modo da poter trasferire i programmi da un computer a un altro senza riscriverli. Tradurre, però, un programma da un linguaggio di programmazione a un altro è un processo difficile, che viene fatto raramente. Pertanto, sembra che le possibilità di impiego, per un numero così elevato di linguaggi di programmazione, siano limitate.

Diversamente dai linguaggi umani, i linguaggi di programmazione vengono creati per scopi specifici. Per esempio, alcuni rendono particolarmente facile la rappresentazione di problemi in un particolare ambito: alcuni linguaggi sono specializzati nell'elaborazione di database; altri nei programmi di "intelligenza artificiale", che tentano di dedurre fatti nuovi da una determinata base di conoscenza; altri linguaggi ancora sono specializzati nella programmazione multimediale. Il linguaggio Pascal fu mantenuto deliberatamente semplice, perché progettato per impieghi didattici. Il linguaggio C fu sviluppato per garantire una sua efficiente traduzione nel veloce linguaggio macchina, con un minimo di attività richiesta per la supervisione. Il linguaggio C++ prende origine dal C, a cui aggiunge nuove caratteristiche per la programmazione

orientata agli oggetti. Il linguaggio Java fu progettato per rendere disponibili programmi nella rete Internet in modo sicuro.

La versione iniziale del linguaggio C fu progettata intorno al 1972. Poiché i progettisti di diversi compilatori aggiunsero al linguaggio caratteristiche tra loro incompatibili, da esso germogliarono in pratica vari dialetti: alcune istruzioni di programmazione erano comprese da un compilatore, ma rifiutate da un altro. Queste divergenze rappresentano un grande fastidio per un programmatore che voglia trasferire codice da un computer a un altro, quindi si fecero grandi sforzi per appianare le differenze e per giungere a una versione standard di C. La fase di progettazione terminò nel 1989, con il completamento dello Standard ANSI (American National Standards Institute). Nel frattempo, Bjarne Stroustrup di AT&T aggiunse al linguaggio C alcune caratteristiche del linguaggio Simula, un linguaggio orientato agli oggetti e progettato per eseguire simulazioni: il linguaggio risultante fu chiamato C++. Dal 1985 a oggi, il linguaggio C++ è cresciuto in seguito all'aggiunta di molti elementi e un processo di standardizzazione venne concluso nel 1988: si è diffuso enormemente, perché i programmatori potevano prendere codice C esistente e convertirlo in C++ apportando soltanto lievi modifiche. Per conservare la compatibilità con il codice esistente, ciascuna innovazione del C++ ha dovuto svilupparsi attorno ai costrutti precedenti, producendo un



James Gosling, progettista del linguaggio di programmazione Java

linguaggio potente, ma piuttosto goffo da usare.

Nel 1995, Java fu progettato da James Gosling per essere concettualmente più semplice e più coerente rispetto al linguaggio C++, sia pure conservando la sintassi che era familiare a milioni di programmatore C e C++. Il linguaggio Java ebbe subito grande successo, non soltanto perché eliminava molti degli aspetti controversi di C++, ma anche per la potenza e completezza della libreria.

In ogni caso, l'evoluzione dei linguaggi di programmazione non si è certamente conclusa e lo stesso linguaggio Java non è in grado di gestire bene alcuni aspetti della programmazione. Si stanno sempre più diffondendo computer dotati di più processori ed è veramente difficile scrivere programmi Java concorren-

*ti*, che usino, cioè, un'architettura multi-processore in modo corretto ed efficiente. Come avete poi visto in questo capitolo, in Java è piuttosto noioso gestire piccoli blocchi di codice, come quello che abbiamo usato per "misurare" un oggetto: occorre progettare un'interfaccia dotata di un unico metodo e, in aggiunta, fornire una classe che la implementa. Nei linguaggi di programmazione *fuzionali*, invece, è possibile manipolare funzioni allo stesso modo degli oggetti e la soluzione di un simile problema diventa molto più semplice.

Ad esempio, nel linguaggio Scala, che presenta caratteristiche ibride tra la programmazione funzionale e quella orientata agli oggetti, si può semplicemente costruire un oggetto di tipo `DataSet` fornendo come parametro una funzione, senza dover usare un'interfaccia:

```
data = new DataSet((x : Rectangle)
=> x.getWidth() * x.getHeight())
```

Si potrebbe migliorare il linguaggio Java per la programmazione concorrente o funzionale? Sono stati

fatti alcuni tentativi, non particolarmente incoraggianti, per via di complesse interazioni con caratteristiche esistenti del linguaggio. Alcuni nuovi linguaggi, come Scala, possono essere eseguiti dalla stessa macchina virtuale usata da Java e possono facilmente invocare codice Java esistente. Nessuno può dire, oggi, quali linguaggi di programmazione avranno maggior successo negli anni a venire, ma la maggior parte degli sviluppatori di software deve prepararsi a lavorare con più linguaggi nel corso della carriera.

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 sono, in realtà, composti da una coppia di eventi di pressione e di rilascio di un pulsante. 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)
    {
        Azione da compiere a ogni click
    }

    // 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) {}
}
```

Tuttavia, questa è una procedura noiosa e, fortunatamente, qualche anima buona ha creato una 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)
    {
        Azione da compiere a ogni click
    }
}
```

Consultate il Capitolo 10 per maggiori informazioni sul procedimento di estensione delle classi.

## Riepilogo degli obiettivi di apprendimento

### Per rendere disponibile un servizio a più classi si usano le interfacce

- In Java, la dichiarazione di un'interfaccia contiene metodi, ma non il relativo codice.
- Diversamente da una classe, un'interfaccia non fornisce alcuna implementazione.
- Per indicare che una classe realizza 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

- Potete effettuare conversioni dal tipo di una classe al tipo di un'interfaccia che sia realizzata dalla classe.
- Per convertire un riferimento a interfaccia in un riferimento a classe serve un *cast*.

### Polimorfismo e ricerca dinamica dei metodi

- Quando la macchina virtuale invoca un metodo di esemplare, usa il metodo della classe che definisce il parametro implicito: si parla di ricerca dinamica del metodo.
- Il polimorfismo è la capacità di gestire in modo omogeneo oggetti aventi comportamenti diversi.

### Come si usano le 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 molto utilizzate per classi con scopo molto limitato, che non hanno bisogno di essere visibili in altre porzioni del programma.

### Gli oggetti semplificati (*mock*) vengono usati nel collaudo

- Un oggetto *mock* fornisce gli stessi servizi di un altro oggetto, in modo semplificato.
- La classe *mock* e la corrispondente classe completa realizzano la medesima interfaccia.

### Eventi e loro ricevitori caratterizzano 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 appartiene a una classe progettata dal programmatore dell'applicazione e i suoi metodi descrivono le azioni da compiere in risposta a un 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 connettete un ActionListener a ogni pulsante.

### I ricevitori di eventi si realizzano mediante classi interne

- I metodi di una classe interna possono accedere alle variabili locali e di esemplare dell'ambito di visibilità circostante.
- Le variabili locali a cui si accede da una classe interna devono essere dichiarate `final`.

### 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 del mouse vanno specificate mediante classi che realizzano l'interfaccia ActionListener.

### Temporizzatori

- 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 disegnate dal metodo paintComponent.

### Eventi del mouse complessi

- Per catturare gli eventi del mouse usate un ricevitore di eventi del mouse (*mouse listener*).

## Classi, oggetti e metodi presentati nel capitolo

|                               |                              |
|-------------------------------|------------------------------|
| java.awt.Component            | java.awt.event.MouseListener |
| addMouseListener              | mouseClicked                 |
| repaint                       | mouseEntered                 |
| setPreferredSize              | mouseExited                  |
| java.awt.Container            | mousePressed                 |
| add                           | mouseReleased                |
| java.awt.Dimension            | javax.swing.AbstractButton   |
| java.awt.Rectangle            | addActionListener            |
| setLocation                   | javax.swing.JButton          |
| java.awt.event.ActionListener | javax.swing.JLabel           |
| actionPerformed               | javax.swing.JPanel           |
| java.awt.event.MouseEvent     | javax.swing.Timer            |
| getX                          | start                        |
| getY                          | stop                         |

## Esercizi di ripasso

- \* **Esercizio R9.1.** Supponete che C sia una classe che realizza le interfacce I e J. Quali dei seguenti assegnamenti richiede un cast?

```
C c = ...;
I i = ...;
J j = ...;
```

- c = i;
- j = c;
- i = j;

- \* **Esercizio R9.2.** Supponete che C sia una classe che realizza le interfacce I e J e che i sia dichiarata e inizializzata in questo modo:

```
I i = new C();
```

Quali dei seguenti enunciati lancia un'eccezione?

- C c = (C) i;

- b. `J j = (J) i;`
- c. `i = (I) null;`

- \* **Esercizio R9.3.** Supponete che `Sandwich` sia una classe che realizza l'interfaccia `Edible` e che siano date queste dichiarazioni e inizializzazioni 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;`

- \*\* **Esercizio R9.4.** In che modo un cast come `(BankAccount) x` è diverso da un cast fra valori numerici, come `(int) x`?

- \*\* **Esercizio R9.5.** Le classi `Rectangle2D.Double`, `Ellipse2D.Double` e `Line2D.Double` realizzano l'interfaccia `Shape`. La classe `Graphics2D` dipende dall'interfaccia `Shape`, ma non dalle classi che descrivono i rettangoli, le ellissi e le linee. Tracciate uno schema UML che illustri questa situazione.

- \*\* **Esercizio R9.6.** Supponete che `r` contenga un riferimento a `new Rectangle(5, 10, 20, 30)`. Quali di questi assegnamenti sono validi? Controllate la documentazione di libreria per verificare quali interfacce siano realizzate 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;`

- \*\* **Esercizio R9.7.** Classi come `Rectangle2D.Double`, `Ellipse2D.Double` e `Line2D.Double` realizzano l'interfaccia `Shape`, la quale ha un metodo

```
Rectangle getBounds()
```

che restituisce il rettangolo circoscritto alla forma. Esaminate l'invocazione:

```
Shape s = ...;
Rectangle r = s.getBounds();
```

e spiegate perché questo è un esempio di polimorfismo.

- \*\*\* **Esercizio R9.8.** In Java, un'invocazione di metodo come `x.f()` usa la ricerca dinamica, perché il metodo giusto da invocare dipende dal tipo di oggetto a cui si riferisce `x`. Fornite due esempi di invocazioni di metodo che non usano la ricerca dinamica in Java.

- \*\* **Esercizio R9.9.** Supponete di dover elaborare un array di impiegati per trovarne il salario medio e massimo. Spiegate cosa dovete fare per usare la prima implementazione della classe `DataSet`, vista nel Paragrafo 9.1, che elabora oggetti di tipo `Measurable`. Cosa dovete fare per usare la seconda implementazione, vista nel Paragrafo 9.4? Quale soluzione è più semplice?
- \*\*\* **Esercizio R9.10.** Cosa succede se aggiungete un oggetto di tipo `String` alla prima implementazione della classe `DataSet`, vista nel Paragrafo 9.1? E con la seconda implementazione di `DataSet`, vista nel Paragrafo 9.4, che usa la classe `RectangleMeasurer`?
- \* **Esercizio R9.11.** Come riorganizzereste il programma di prova `DataSetTester3` se aveste bisogno di rendere `RectangleMeasurer` una classe di primo livello (cioè una classe non interna)?
- \*\* **Esercizio R9.12.** Cos'è un oggetto *callback*? Riuscite a immaginare un altro utile *callback* per la classe `DataSet`? Suggerimento: Esercizio P9.12.
- \*\* **Esercizio R9.13.** 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;
        ...
    }
}

```

- \*\* **Esercizio R9.14.** Cosa succede quando una classe interna cerca di accedere a una variabile locale non `final`? Provate e date una spiegazione di ciò che osservate.
- \*\*\*G **Esercizio R9.15.** Come riorganizzereste il programma `InvestmentViewer1` se aveste bisogno di rendere `AddInterestListener` una classe di primo livello (cioè una classe non interna)?
- \*G **Esercizio R9.16.** Cos'è un oggetto che rappresenta un evento? Cos'è una sorgente di evento? Cos'è un ricevitore di eventi?
- \*G **Esercizio R9.17.** 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?
- \*G **Esercizio R9.18.** Spiegate la differenza fra un oggetto di tipo `ActionEvent` e un oggetto di tipo `MouseEvent`.
- \*\*G **Esercizio R9.19.** Spiegate perché l'interfaccia `ActionListener` ha un solo metodo, mentre `MouseListener` ne ha cinque.

- \*\*G Esercizio R9.20.** Una classe può essere un'origine di evento per più tipi di evento? Se sì, fornite un esempio.
- \*\*G Esercizio R9.21.** Quali informazioni contiene un oggetto di tipo `ActionEvent`? Quali informazioni aggiuntive sono contenute in un oggetto di tipo `MouseEvent`?
- \*\*\*G Esercizio R9.22.** Perché utilizziamo classi interne per i ricevitori di eventi? Se Java non consentisse di utilizzare le classi interne, potremmo ugualmente realizzare ricevitori di eventi? In quale modo?
- \*\*G Esercizio R9.23.** Qual è la differenza fra i metodi `paintComponent` e `repaint`.
- \*G Esercizio R9.24.** Qual è la differenza fra un frame e un pannello?

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

# 10

## Ereditarietà

### Obiettivi del capitolo

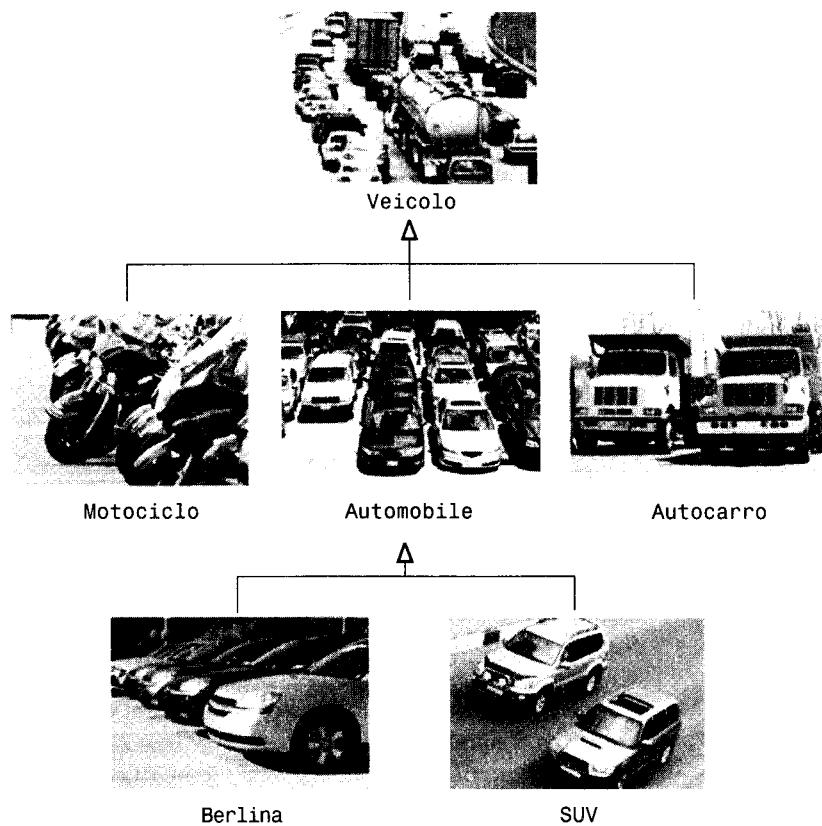
- Imparare l'ereditarietà
- Capire come ereditare e come sovrascrivere metodi di superclasse
- Saper invocare costruttori della superclasse
- Conoscere il controllo d'accesso protetto e di pacchetto
- Capire il concetto di superclasse comune, `Object`, e saper sovrascrivere i suoi metodi `toString` e `equals`
- Usare l'ereditarietà per personalizzare le interfacce utente

In questo capitolo viene presentato il fondamentale concetto di ereditarietà: si possono creare classi specializzate che ereditano il comportamento di classi più generiche. Imparerete a realizzare l'ereditarietà in Java e studierete come utilizzare la classe `Object`, la classe più generica della gerarchia di ereditarietà.

## 10.1 Gerarchie di ereditarietà

Frequentemente nel mondo reale i concetti si classificano secondo *gerarchie*, che vengono spesso rappresentate mediante alberi, con i concetti più generali vicini alla radice della gerarchia e quelli più specializzati nelle diramazioni. La Figura 1 mostra un tipico esempio.

**Figura 1**  
Una gerarchia di tipi di veicolo.



Insiemi di classi possono dar luogo a gerarchie di ereditarietà complesse.

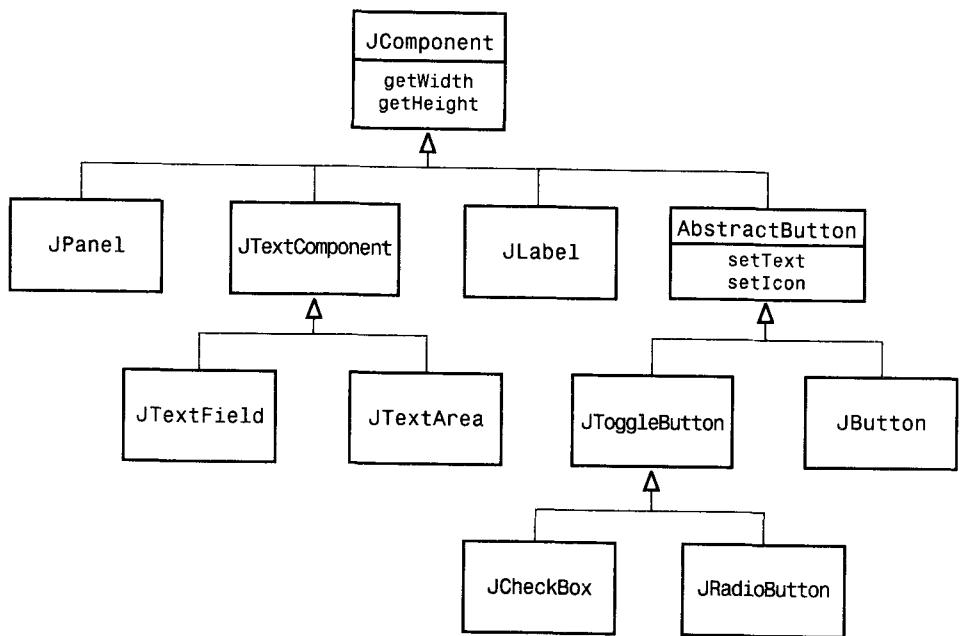
In Java, è altrettanto comune raggruppare le classi in complesse *gerarchie di ereditarietà*. Le classi che rappresentano i concetti più generali sono vicine alla radice dell'albero, mentre quelle più specializzate si trovano nelle diramazioni. La Figura 2 mostra una parte della gerarchia dei componenti per l'interfaccia utente grafica Swing di Java.

Dobbiamo ora introdurre qualche ulteriore elemento di terminologia per poter esprimere le relazioni esistenti tra classi all'interno di una gerarchia di ereditarietà. La classe più generale viene detta *superclasse*, mentre una classe più specializzata, che eredita dalla superclasse, viene detta *sottoclasse*. Nel nostro esempio, JPanel è una sottoclasse di JComponent.

Nella Figura 2 abbiamo usato la notazione UML relativa all'ereditarietà: in un diagramma di classi, si rappresenta l'ereditarietà mediante una freccia a tratto continuo che termina con un triangolo diretto verso la superclasse.

**Figura 2**

Una parte della gerarchia dei componenti per l'interfaccia utente grafica Swing.



Quando progettate una gerarchia di classi, domandatevi quali caratteristiche e comportamenti sono comuni a tutte le classi che state progettando: tali proprietà comuni vanno raccolte in una superclasse. Ad esempio, tutti i componenti dell'interfaccia utente grafica hanno una larghezza e un'altezza: i metodi `getWidth` e `getHeight` della classe **JComponent** restituiscono tali dimensioni. Nelle sottoclassi si possono trovare proprietà più specializzate; per esempio, i pulsanti possono usare testo e icone come etichette. La classe **AbstractButton**, ma non la superclasse **JComponent**, ha i metodi per impostare testo e icone per i pulsanti, oltre alle variabili di esemplare per memorizzarli. Le singole classi per i pulsanti (come **JButton**, **JRadioButton** e **JCheckBox**) ereditano queste proprietà e, di fatto, la classe **AbstractButton** è stata creata per rappresentare le funzionalità che accomunano tali pulsanti.

Nel nostro studio relativo ai concetti di ereditarietà useremo un esempio di gerarchia più semplice. Considerate un banca che offre ai suoi clienti due tipi di conto:

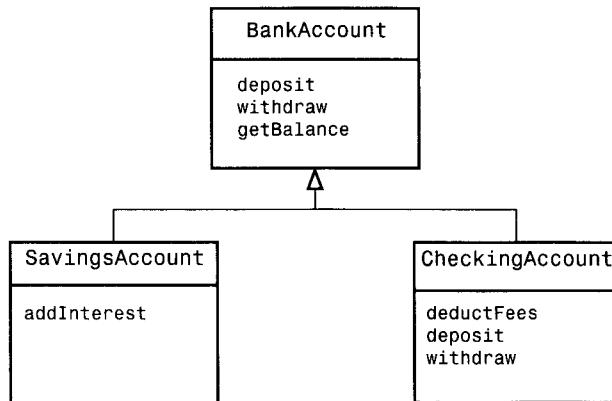
1. Il conto corrente (*checking account*) non offre interessi, concede un numero limitato di transazioni mensili gratuite e addebita una commissione per ciascuna transazione aggiuntiva.
2. Il conto di risparmio (*savings account*) frutta interessi mensili (nella nostra realizzazione si calcola l'interesse sul saldo dell'ultimo giorno del mese, ma è una procedura poco realistica, perché generalmente le banche usano il saldo medio o il saldo minimo giornaliero; l'Esercizio P10.1 vi chiederà di realizzare questo miglioramento).

La gerarchia di ereditarietà è riportata nella Figura 3. L'Esercizio P10.2 vi chiederà di aggiungere un'altra classe a questa gerarchia.

Proviamo ora a determinare il comportamento di queste classi. Tutti i conti bancari mettono a disposizione il metodo `getBalance`, che restituisce semplicemente il saldo

**Figura 3**

Gerarchia di ereditarietà per classi di conti bancari.



attuale. Tutti forniscono anche i metodi `deposit` e `withdraw`, sia pure con dettagli diversi nella realizzazione: per esempio, un conto corrente deve registrare il numero delle transazioni, per poter poi calcolare le commissioni da addebitare.

Il conto corrente ha bisogno di un metodo `deductFees` per addebitare le commissioni mensili e azzerare il contatore delle transazioni. Pertanto, bisogna sovrascrivere i metodi `deposit` e `withdraw`, perché contino le transazioni.

I conti di risparmio hanno bisogno di un metodo `addInterest`, per calcolare e accreditare gli interessi.

Riassumendo: le sottoclassi mettono a disposizione tutti i metodi della loro superclasse, ma la realizzazione di tali metodi può essere modificata per soddisfare specifici requisiti di ciascuna sottoclasse. Inoltre, le sottoclassi possono liberamente aggiungere nuovi metodi.



### Auto-valutazione

1. A cosa serve la classe `JTextComponent` di Figura 2?
2. Perché non inseriamo il metodo `addInterest` nella classe `BankAccount`?

## 10.2 Realizzare sottoclassi

L'ereditarietà è uno strumento per estendere classi esistenti aggiungendo metodi e variabili di esemplare.

Una sottoclasse eredita i metodi della sua superclasse.

In questo paragrafo inizieremo a costruire la gerarchia di ereditarietà per le classi che rappresentano conti bancari: per farlo, imparerete a progettare una sottoclasse a partire da una superclasse, iniziando con la classe `SavingsAccount`. Ecco la sintassi per dichiarare la classe:

```

public class SavingsAccount extends BankAccount
{
    nuove variabili di esemplare
    nuovi metodi
}
  
```

Nella dichiarazione della classe `SavingsAccount` si specificano soltanto i metodi nuovi e le variabili di esemplare nuove. Tutti i metodi della classe `BankAccount` vengono *ereditati automaticamente* dalla classe `SavingsAccount`.

Per esempio, il metodo `deposit` si applica automaticamente ai conti di risparmio:

```
SavingsAccount collegeFund = new SavingsAccount(10);
    // un conto di risparmio con tasso di interesse del 10%
collegeFund.deposit(500);
    // è corretto usare un metodo di BankAccount
    // con un oggetto di tipo SavingsAccount
```

Proviamo a vedere in che modo gli oggetti di tipo “conto di risparmio” differiscono da quelli di tipo `BankAccount`: imposteremo un tasso di interesse nel costruttore e ci servirà un metodo per accreditare periodicamente gli interessi. In pratica, in aggiunta ai tre metodi che si possono applicare a ciascun conto bancario, esiste un metodo addizionale, `addInterest`: questo nuovo metodo e la nuova variabile di esemplare vanno dichiarati nella sottoclassa.

```
public class SavingsAccount extends BankAccount
{
    private double interestRate;

    public SavingsAccount(double rate)
    {
        realizzazione del costruttore
    }

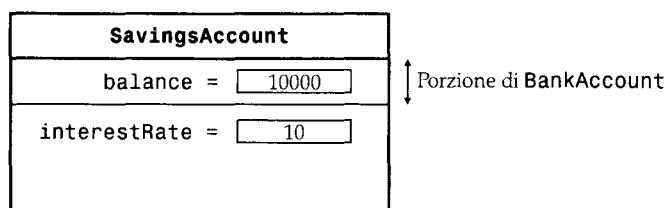
    public void addInterest()
    {
        realizzazione del metodo
    }
}
```

Le variabili di esemplare dichiarate nella superclasse sono presenti anche negli oggetti di una sottoclassa.

Un esemplare di una sottoclassa possiede automaticamente tutte le variabili di esemplare dichiarate nella superclasse. Ad esempio, un oggetto di tipo `SavingsAccount` ha la variabile di esemplare `balance`, che è stata dichiarata nella classe `BankAccount`.

Ogni nuova variabile di esemplare che venga dichiarata in una sottoclassa è, invece, presente soltanto negli esemplari di tale sottoclassa. Ad esempio, ogni oggetto di tipo `SavingsAccount` ha la variabile di esemplare `interestRate`. La Figura 4 mostra la struttura interna di un oggetto di tipo `SavingsAccount`.

**Figura 4**  
La struttura di un esemplare di sottoclassa.



Successivamente, dovete realizzare il nuovo metodo `addInterest`, che calcola gli interessi maturati in base al saldo attuale, per poi versarli nel conto.

```
public class SavingsAccount extends BankAccount
{
    private double interestRate;
```

```

public SavingsAccount(double rate)
{
    interestRate = rate;
}

public void addInterest()
{
    double interest = getBalance() * interestRate / 100;
    deposit(interest);
}
}

```

 Una sottoclasse non ha accesso alle variabili private della sua superclasse.

Potreste, a questo punto, chiedervi quale sia il motivo per cui il metodo `addInterest` invoca i metodi `getBalance` e `deposit` invece di aggiornare direttamente la variabile `balance` della superclasse. Si tratta di una conseguenza dell'*incapsulamento*: la variabile `balance` è stata dichiarata `private` nella classe `BankAccount` e il metodo `addInterest` è dichiarato nella classe `SavingsAccount`, per cui non ha accesso a una variabile privata di un'altra classe.

Notate come il metodo `addInterest` invochi i metodi ereditati `getBalance` e `deposit` senza specificare un parametro implicito: ciò significa che le invocazioni si applicano al parametro implicito del metodo `addInterest`.

In altre parole, gli enunciati presenti nel metodo `addInterest` sono un'abbreviazione degli enunciati seguenti:

```

double interest = this.getBalance() * this.interestRate / 100;
this.deposit(interest);

```

## Sintassi di Java

### 10.1 Ereditarietà

#### Sintassi

```

class NomeSottoclasse extends NomeSuperclasse
{
    variabili di esemplare
    metodi
}

```

#### Esempio

Dichiarazione delle variabili di esemplare che vengono aggiunte alla sottoclasse.

```

Sottoclasse
/
public class SavingsAccount extends BankAccount
{
    private double interestRate;
    ...
}

```

Dichiarazione dei metodi che sono specifici della sottoclasse.

```

    public void addInterest()
    {
        double interest = getBalance() * interestRate / 100;
        deposit(interest);
    }
}

```

La parola riservata `extends` caratterizza l'ereditarietà.

Questo completa la realizzazione della classe `SavingsAccount`, di cui trovate il codice nel seguito.

Ereditare da una classe  
non è come realizzare un'interfaccia:  
la sottoclasse eredita  
il comportamento della propria  
superclasse.

### File ch10/accounts/SavingsAccount.java

```
/**
 * Un conto bancario che remunera interessi con un tasso fisso.
 */
public class SavingsAccount extends BankAccount
{
    private double interestRate;

    /**
     * Costruisce un conto bancario con un tasso d'interesse assegnato.
     * @param rate il tasso d'interesse
     */
    public SavingsAccount(double rate)
    {
        interestRate = rate;
    }

    /**
     * Aggiunge al conto bancario gli interessi maturati.
     */
    public void addInterest()
    {
        double interest = getBalance() * interestRate / 100;
        deposit(interest);
    }
}
```

### Auto-valutazione

3. Quali sono le variabili di esemplare presenti in un oggetto di tipo `SavingsAccount`?
4. Elicitate quattro metodi che possono essere invocati con un oggetto di tipo `SavingsAccount`.
5. Se la classe `Manager` estende la classe `Employee`, quale classe ha il ruolo di superclasse e quale, invece, ha il ruolo di sottoclasse?

## Errori comuni 10.1

### Confondere superclassi e sottoclassi

Se confrontate un oggetto di tipo `SavingsAccount` con uno di tipo `BankAccount`, potete notare che:

- La parola riservata `extends` suggerisce che l'oggetto `SavingsAccount` sia una versione estesa di un oggetto `BankAccount`.

- L'oggetto `SavingsAccount` è più grande, perché ha una variabile di esemplare aggiuntiva, `interestRate`.
- L'oggetto `SavingsAccount` è più abile, perché ha un metodo aggiuntivo, `addInterest`.

Sembra che l'oggetto `SavingsAccount` sia superiore all'altro sotto tutti i punti di vista; quindi, perché `SavingsAccount` viene detta *sottoclasse*, mentre `BankAccount` è la *superclasse*?

La terminologia *super/sotto* deriva dalla teoria degli insiemi. Considerate l'insieme di tutti i conti bancari: non tutti sono oggetti di tipo `SavingsAccount`, alcuni sono conti bancari di tipo diverso. Pertanto, l'insieme degli oggetti di tipo `SavingsAccount` è un *sottoinsieme* dell'insieme formato dagli oggetti di tipo `BankAccount`, che, per contro, rappresenta un suo *superinsieme*. Gli oggetti del sottoinsieme, maggiormente specializzati, hanno uno stato più ricco e maggiori capacità.



## Errori comuni 10.2

### Mettere in ombra variabili di esemplare

Una sottoclassa non ha accesso alle variabili di esemplare private della superclasse. Ad esempio, i metodi della classe `SavingsAccount` non possono accedere alla variabile di esemplare `balance`:

```
public class SavingsAccount extends BankAccount
{
    public void addInterest()
    {
        double interest = getBalance() * interestRate / 100;
        balance = balance + interest; // ERRORE
    }
    ...
}
```

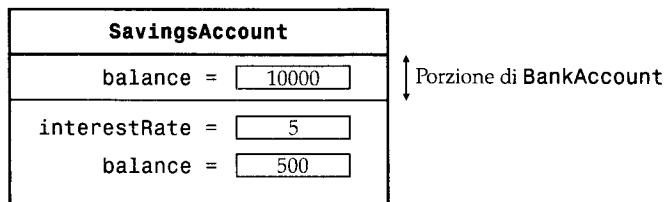
È un frequente errore da principianti “risolvere” questo problema aggiungendo *un'altra* variabile di esemplare avente lo stesso nome.

```
public class SavingsAccount extends BankAccount
{
    private double balance; // NON FATELO
    ...
    public void addInterest()
    {
        double interest = getBalance() * interestRate / 100;
        balance = balance + interest; // Compila ma non aggiorna il saldo giusto
    }
}
```

Ora il metodo `deposit` viene certamente compilato, ma non aggiorna il saldo giusto! Un tale oggetto `SavingsAccount` ha due variabili di esemplare, entrambe di nome `balance` (osservate la Figura 5). Il metodo `getBalance` della superclasse ne usa una, mentre il metodo `addInterest` della sottoclassa aggiorna l'altra.

**Figura 5**

Mettere in ombra variabili di esemplare.



## 10.3 Sovrascrivere metodi

Una sottoclasse può ereditare un metodo dalla superclasse oppure sovrascriverlo, fornendone una realizzazione alternativa.

Passiamo ora a progettare la classe **CheckingAccount**, per vedere un esempio di metodo sovrascritto. Ricordate che la classe **BankAccount** ha tre metodi:

```
public class BankAccount
{
    ...
    public void deposit(double amount) { ... }
    public void withdraw(double amount) { ... }
    public double getBalance() { ... }
}
```

La classe **CheckingAccount** dichiara questi metodi:

```
public class CheckingAccount extends BankAccount
{
    ...
    public void deposit(double amount) { ... } // metodo ridefinito
    public void withdraw(double amount) { ... } // metodo ridefinito
    public void deductFees() { ... }
}
```

I metodi **deposit** e **withdraw** della classe **CheckingAccount** sovrascrivono, rispettivamente, i metodi **deposit** e **withdraw** della classe **BankAccount**, per poter gestire le commissioni relative alle transazioni effettuate. Il metodo **deductFees**, invece, non sovrascrive alcun altro metodo, e il metodo **getBalance** non viene sovrascritto.

Realizziamo nella classe **CheckingAccount** il metodo **deposit**, che incrementa il conteggio delle transazioni e versa il denaro:

```
public class CheckingAccount extends BankAccount
{
    ...
    public void deposit(double amount)
    {
        transactionCount++;
        // poi aggiunge amount al saldo
        ...
    }
}
```

Qui abbiamo un problema: non possiamo sommare semplicemente `amount` a `balance`:

```
public class CheckingAccount extends BankAccount
{
    ...
    public void deposit(double amount)
    {
        transactionCount++;
        // poi aggiunge amount al saldo
        balance = balance + amount; // ERRORE
    }
}
```

Sebbene ciascun oggetto di tipo `CheckingAccount` abbia una propria variabile di esemplare `balance`, si tratta di una variabile *privata* della superclasse `BankAccount` e i metodi della sottoclasse non hanno maggiori diritti di accesso ai dati privati della superclasse di quanti ne abbia qualsiasi altro metodo. Se volete modificare una variabile di esemplare privata di una superclasse, dovete usare un metodo pubblico della superclasse stessa.

Come possiamo sommare al saldo l'importo di un versamento, usando l'interfaccia pubblica della classe `BankAccount`? Esiste un metodo che si presta perfettamente a questo scopo, il metodo `deposit` della classe `BankAccount`, dobbiamo, quindi, invocare `deposit` su qualche oggetto. Quale? Il conto corrente in cui si deposita il denaro, ovvero il parametro implicito del metodo `deposit` della classe `CheckingAccount`. Per invocare un altro metodo con il medesimo parametro implicito, non occorre specificare il parametro, basta scrivere il nome del metodo, in questo modo:

```
public class CheckingAccount extends BankAccount
{
    ...
    public void deposit(double amount)
    {
        transactionCount++;
        // poi aggiunge amount al saldo
        deposit(amount); // ancora incompleto
    }
}
```

Questo codice non funziona ancora. Il compilatore interpreta l'enunciato:

```
deposit(amount);
```

come se fosse:

```
this.deposit(amount);
```

Il parametro `this` è di tipo `CheckingAccount` e nella classe `CheckingAccount` esiste un metodo chiamato `deposit`. Viene, quindi, invocato quel metodo, ma si tratta proprio del metodo che stiamo scrivendo! Il metodo invocherà se stesso ripetutamente, facendo cadere il programma in una ricorsione infinita (argomento che verrà discusso nel Capitolo 12).

## Sintassi di Java

### 10.2 Invocare un metodo della superclasse

#### Sintassi

```
super.nomeMetodo(parametri);
```

#### Esempio

```
public void deposit(double amount)
{
    transactionCount++;
    super.deposit(amount);
}
```

Invoca il metodo della superclasse invece del proprio metodo.

Se vi dimenticate super, questo metodo invoca se stesso.

Per invocare un metodo della superclasse usate la parola riservata **super**.

Dobbiamo, invece, specificare che intendiamo invocare il metodo **deposit** della superclasse. Per questo scopo esiste una parola riservata speciale, **super**:

```
public class CheckingAccount extends BankAccount
{
    ...
    public void deposit(double amount)
    {
        transactionCount++;
        // poi aggiunge amount al saldo
        super.deposit(amount);
    }
}
```

Questa versione del metodo **deposit** è corretta: per versare denaro in un conto corrente, aggiornate il conteggio delle transazioni e, poi, invocate il metodo **deposit** della superclasse.

Anche gli altri metodi della classe **CheckingAccount** invocano un metodo della superclasse.

```
public class CheckingAccount extends BankAccount
{
    private static final int FREE_TRANSACTIONS = 3;
    private static final double TRANSACTION_FEE = 2.0;

    private int transactionCount;
    ...
    public void withdraw(double amount)
    {
        transactionCount++;
        // poi sottrae amount dal saldo
        super.withdraw(amount);
    }

    public void deductFees()
```

```

    {
        if (transactionCount > FREE_TRANSACTIONS)
        {
            double fees = TRANSACTION_FEE
                * (transactionCount - FREE_TRANSACTIONS);
            super.withdraw(fees);
        }
        transactionCount = 0;
    }
    ...
}

```



## Auto-valutazione

6. Catalogate i metodi della classe `SavingsAccount` come ereditati, nuovi o ridefiniti.
7. Perché il metodo `withdraw` della classe `CheckingAccount` invoca `super.withdraw`?
8. Perché il metodo `deductFees` azzera il conteggio delle transazioni effettuate?



## Errori comuni 10.3

### Sovrascrivere accidentalmente

Ricordate, dal Paragrafo 2.4, che due metodi possono avere lo stesso nome, ma devono avere parametri *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 il proprio codice: il compilatore Java li considera completamente incorrelati e si dice che il nome `println` è *sovaccarico* (“overloaded”). Si tratta di un fenomeno diverso dalla sovrascrittura, dove un metodo di sottoclasse fornisce una diversa realizzazione 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 sovaccarico. In questo esempio

```

public class CheckingAccount extends BankAccount
{
    ...
    public void deposit(int amount) // ERRORE: dovrebbe essere double
    {
        ...
    }
}

```

il compilatore non protesta: pensa che vogliate fornire un metodo `deposit` per valori di tipo `int`, ereditando l'altro metodo `deposit` per valori di tipo `double`.

Quando sovrascrivete un metodo, controllate con cura che i tipi dei parametri siano esatti.

## Errori comuni 10.4

### Sbagliare nell'invocare metodi della superclasse

Quando si estendono le funzionalità di un metodo di una superclasse, un errore comune consiste nel dimenticarsi il qualificatore `super`. Ad esempio, per prelevare denaro da un conto corrente, aggiornate il conteggio delle transazioni e prelevate l'importo:

```
public void withdraw(double amount)
{
    transactionCount++;
    withdraw(amount);
    // Errore: dovrebbe essere super.withdraw(amount)
}
```

Qui, `withdraw(amount)` si riferisce al metodo `withdraw` applicato al parametro implicito del metodo, che è di tipo `CheckingAccount`: la classe `CheckingAccount` ha un metodo `withdraw`, che viene quindi invocato. Naturalmente, questo provoca l'invocazione dello stesso metodo, che quindi invocherà nuovamente se stesso, una volta dopo l'altra, finché il programma esaurirà la memoria disponibile. Invece, dovete identificare con maggior precisione quale metodo `withdraw` volete invocare.

Un altro errore comune consiste nel dimenticarsi completamente di invocare il metodo della superclasse: la funzionalità della superclasse svanisce misteriosamente.

## 10.4 Costruttori in sottoclassi

In questo paragrafo parliamo della realizzazione dei costruttori in una sottoclasse. Come esempio, dichiariamo un costruttore per impostare il saldo iniziale di un conto corrente, cioè di un oggetto di tipo `CheckingAccount`.

Vogliamo invocare il costruttore di `BankAccount` per impostare l'importo del saldo iniziale. Esiste un'istruzione speciale per invocare il costruttore della superclasse dal costruttore di una sottoclasse: si usa la parola riservata `super`, seguita dai parametri di costruzione fra parentesi.

```
public class CheckingAccount extends BankAccount
{
    public CheckingAccount(double initialBalance)
    {
        // costruisci la superclasse
        super(initialBalance);
        // inizializza il contatore delle transazioni
        transactionCount = 0;
    }
    ...
}
```

**Sintassi**

```
modalitàDiAccesso NomeClasse(TipoDiParametro nomeParametro, ...)  
{  
    super(parametri);  
    ...  
}
```

**Esempio**

Invoca il costruttore della superclasse.

Deve essere il primo enunciato nel costruttore della sottoclasse.

```
public CheckingAccount(double initialBalance)  
{  
    super(initialBalance);  
    transactionCount = 0;  
}
```

Se non c'è questa riga, viene invocato il costruttore della superclasse privo di parametri.

Per invocare il costruttore della superclasse si usa la parola riservata `super` nel primo enunciato del costruttore della sottoclasse.

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.

Il doppio uso della parola riservata `super` è analogo al doppio uso di `this`, visto in Argomenti avanzati 3.1.

Se un costruttore di una sottoclasse non invoca il costruttore della superclasse, la superclasse deve avere un costruttore privo di parametri, che viene utilizzato per inizializzare i dati relativi alla superclasse. Se, invece, tutti i costruttori della superclasse richiedono parametri, il compilatore segnala un errore.

Per esempio, potete realizzare il costruttore di `CheckingAccount` senza invocare il costruttore della superclasse: di conseguenza, viene invocato il costruttore `BankAccount()`, che imposterà il saldo a zero. Naturalmente, il costruttore di `CheckingAccount` dovrà poi fare un versamento esplicito per costituire il saldo iniziale previsto.

Più spesso, tuttavia, i costruttori delle sottoclassi hanno alcuni parametri da passare alla superclasse e altri da usare per inizializzare le proprie variabili di esemplare, dichiarate nella sottoclasse stessa.

**File ch10/accounts/CheckingAccount.java**

```
/**  
 * Un conto corrente che addebita commissioni per le transazioni.  
 */  
public class CheckingAccount extends BankAccount  
{  
    private static final int FREE_TRANSACTIONS = 3;  
    private static final double TRANSACTION_FEE = 2.0;
```

```
private int transactionCount;

/**
 * Costruisce un conto corrente con un saldo iniziale assegnato.
 * @param initialBalance il saldo iniziale
 */
public CheckingAccount(double initialBalance)
{
    // costruisce la superclasse
    super(initialBalance);

    // azzerà il conteggio delle transazioni
    transactionCount = 0;
}

public void deposit(double amount)
{
    transactionCount++;
    // poi aggiunge amount al saldo
    super.deposit(amount);
}

public void withdraw(double amount)
{
    transactionCount++;
    // poi sottrae amount dal saldo
    super.withdraw(amount);
}

/**
 * Sottrae dal saldo le commissioni totalizzate
 * e azzerà il conteggio delle transazioni.
 */
public void deductFees()
{
    if (transactionCount > FREE_TRANSACTIONS)
    {
        double fees = TRANSACTION_FEE
                     * (transactionCount - FREE_TRANSACTIONS);
        super.withdraw(fees);
    }
    transactionCount = 0;
}
```



## Auto-valutazione

9. Perché il costruttore della classe **SavingsAccount** visto nel Paragrafo 10.2 non invoca il costruttore della superclasse?
10. Quando invocate un metodo della superclasse usando la parola riservata **super**, tale invocazione deve necessariamente essere il primo enunciato del metodo della sottoclasse?

## 10.5 Conversione di tipo fra sottoclasse e superclasse

Spesso si ha la necessità di convertire un tipo di sottoclasse nel tipo della sua superclasse: a volte si deve invece effettuare la conversione opposta: questo paragrafo discute tali regole di conversione.

I riferimenti a sottoclassi possono essere convertiti in riferimenti a superclassi.

La classe `SavingsAccount` estende la classe `BankAccount`: in altre parole, un oggetto di tipo `SavingsAccount` è un caso speciale di oggetto di tipo `BankAccount`, quindi un riferimento a un oggetto di tipo `SavingsAccount` può essere convertito in un riferimento di tipo `BankAccount`.

```
SavingsAccount collegeFund = new SavingsAccount(10);
BankAccount anAccount = collegeFund; // si può fare
```

Inoltre, qualsiasi riferimento può essere convertito in `Object`.

```
Object anObject = collegeFund; // si può fare
```

A questo punto, i tre riferimenti memorizzati in `collegeFund`, `anAccount` e `anObject` puntano tutti allo stesso oggetto di tipo `SavingsAccount` (osservate la Figura 6).

Tuttavia, le variabili `anAccount` e `anObject` non conoscono la vera natura dell'oggetto a cui si riferiscono: dato che `anAccount` è una variabile di tipo `BankAccount`, potete invocare con essa i metodi `deposit` e `withdraw`, ma non il metodo `addInterest`, perché non è un metodo della classe `BankAccount`:

```
anAccount.deposit(1000); // va bene
anAccount.addInterest();
// No, non è un metodo di BankAccount, il tipo di anAccount
```

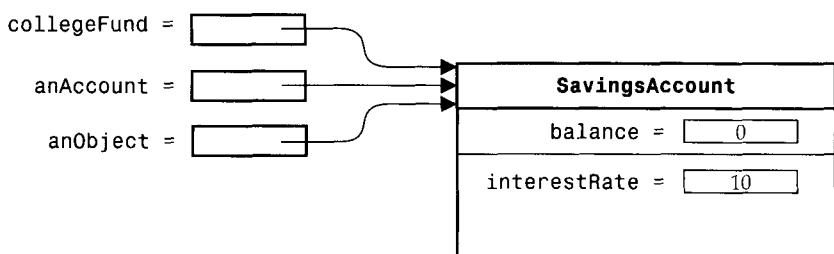
Ovviamente, la variabile `anObject` sa ancora meno: con tale variabile non potete neanche invocare il metodo `deposit`, perché non è un metodo della classe `Object`.

Perché mai qualcuno vorrebbe *intenzionalmente* avere meno informazioni a proposito di un riferimento e memorizzarlo in una variabile il cui tipo è la superclasse dell'oggetto effettivo? Ciò può accadere quando si vuole *riutilizzare codice* che si interessa della superclasse, ignorando i dettagli relativi alla sottoclasse. Ecco un esempio tipico. Considerate un metodo `transfer` che trasferisce denaro da un conto a un altro:

```
public void transfer(double amount, BankAccount other)
{
    withdraw(amount);
    other.deposit(amount);
}
```

**Figura 6**

Variabili di tipo diverso si riferiscono al medesimo oggetto



Potete usare questo metodo per trasferire denaro da un conto bancario a un altro:

```
BankAccount momsAccount = ...;
BankAccount harrysAccount = ...;
momsAccount.transfer(1000, harrysAccount);
```

Potete usare il metodo *anche* per trasferire denaro in un conto corrente di tipo `CheckingAccount`:

```
CheckingAccount harrysChecking = ...;
momsAccount.transfer(1000, harrysChecking);
// è lecito passare una variabile di tipo CheckingAccount
// a un metodo che si aspetta una variabile di tipo BankAccount
```

Il metodo `transfer` si aspetta di ricevere come parametro un riferimento di tipo `BankAccount` e riceve, invece, un riferimento a un oggetto di tipo `CheckingAccount`. Si tratta di una situazione perfettamente lecita. Il metodo `transfer` non sa che, in questo caso, `other` contiene in realtà un riferimento a un oggetto di tipo `CheckingAccount`; tutto ciò che gli interessa sapere è che l'oggetto può compiere l'operazione `deposit` e questo è garantito, perché la variabile `other` è di tipo `BankAccount`.

Più raramente c'è bisogno di effettuare la conversione opposta, dal tipo di una superclasse al tipo di una sua sottoclasse. Ad esempio, può darsi che disponiate di una variabile di tipo `Object`, ma voi sapete che, in realtà, contiene un riferimento a un oggetto di tipo `BankAccount`. In tal caso potete usare un cast esplicito per effettuare la conversione di tipo:

```
BankAccount anAccount = (BankAccount) anObject;
```

Questo cast, però, è piuttosto pericoloso: se vi sbagliate e la variabile `anObject` punta, in realtà, a un oggetto di tipo diverso, privo di parentela con `BankAccount`, viene lanciata un'eccezione.

Come protezione contro i cast errati potete usare l'operatore `instanceof`, che verifica se un oggetto è di un particolare tipo. Ad esempio

```
anObject instanceof BankAccount
```

restituisce `true` se il riferimento contenuto nella variabile `anObject` può essere convertito in un riferimento di tipo `BankAccount`, cosa che accade se `anObject` punta effettivamente a un oggetto di tipo `BankAccount` oppure di una sua sottoclasse, come `SavingsAccount`. Usando l'operatore `instanceof`, si può programmare un cast sicuro in questo modo:

```
if (anObject instanceof BankAccount)
{
    BankAccount anAccount = (BankAccount) anObject;
    ...
}
```

## Auto-valutazione

11. Perché il secondo parametro del metodo `transfer` deve essere di tipo `BankAccount` e non, per esempio, di tipo `SavingsAccount`?
12. Perché non possiamo modificare il secondo parametro del metodo `transfer` in modo che sia di tipo `Object`?

L'operatore `instanceof` verifica  
se un oggetto  
è di un particolare tipo.



## Sintassi di Java

## 10.4 L'operatore instanceof

### Sintassi

oggetto instanceof NomeTipo

### Esempio

Se anObject vale null,  
instanceof restituisce false.

Con questa variabile si possono  
invocare metodi di BankAccount.

Restituisce true se anObject può essere  
convertito in BankAccount.

```
if (anObject instanceof BankAccount)
{
    BankAccount anAccount = (BankAccount) anObject;
    ...
}
```

L'oggetto può appartenere a una  
sottoclasse di BankAccount.

Due riferimenti  
al medesimo oggetto.

## 10.6 Polimorfismo e ereditarietà

In Java, il tipo di una variabile non corrisponde al tipo dell'oggetto a cui essa fa riferimento. Ad esempio, una variabile di tipo `BankAccount` può contenere un riferimento a un oggetto che è veramente di tipo `BankAccount` o a un oggetto di una sua sottoclasse, come `SavingsAccount`. Avete già osservato questo fenomeno nel Capitolo 9 con variabili di tipo interfaccia: una variabile di tipo `Measurable` contiene un riferimento a un oggetto di una classe che realizza l'interfaccia `Measurable`, ad esempio un oggetto di tipo `Coin` oppure un oggetto di una classe completamente diversa.

Cosa accade quando viene invocato un metodo usando una variabile di tipo `BankAccount`? Analizziamo questo esempio:

```
BankAccount anAccount = new CheckingAccount();
anAccount.deposit(1000);
```

Quale metodo `deposit` viene invocato? La variabile `anAccount` è di tipo `BankAccount`, per cui potrebbe sembrare che venga invocato `BankAccount.deposit`. D'altra parte, la classe `CheckingAccount` ha il proprio metodo `deposit`, che aggiorna il conteggio delle transazioni. Il riferimento memorizzato nella variabile `anAccount` punta, in realtà, a un esemplare della sottoclasse `CheckingAccount`, per cui sarebbe più giusto che venisse invocato il metodo `CheckingAccount.deposit`.

Java determina il metodo da invocare mediante la *ricerca dinamica del metodo* (“dynamic method lookup”). Il metodo da invocare è sempre deciso in base al tipo reale dell'oggetto, non in base al tipo della variabile: se l'oggetto è, in realtà, di tipo `CheckingAccount`, viene invocato il metodo `CheckingAccount.deposit`, indipendentemente dal fatto che il riferimento all'oggetto sia memorizzato in una variabile di tipo `BankAccount`.

Torniamo al metodo `transfer`:

```
public void transfer(double amount, BankAccount other)
{
```

Quando la macchina virtuale invoca  
un metodo di esemplare, lo cerca  
nella classe corrispondente  
al parametro implicito, mediante  
ricerca dinamica del metodo.

```
    withdraw(amount);
    other.deposit(amount);
}
```

Immaginate di invocare:

```
anAccount.transfer(1000, anotherAccount);
```

Come risultato, si avrebbero due invocazioni di metodo:

```
anAccount.withdraw(1000);
anotherAccount.deposit(1000);
```

In relazione ai reali tipi degli oggetti i cui riferimenti sono memorizzati nelle variabili `anAccount` e `anotherAccount`, vengono invocate versioni diverse dei metodi `withdraw` e `deposit`. È un esempio di *polimorfismo*, che, come già detto nel Capitolo 9, è la funzionalità che consente di trattare in modo omogeneo oggetti che hanno comportamenti diversi.

Se esaminate la realizzazione del metodo `transfer`, potrebbe non essere di immediata comprensione il fatto che la prima invocazione del metodo

```
withdraw(amount);
```

dipende dal tipo di un oggetto. Dovete però ricordare che tale invocazione è un'abbreviazione per questa:

```
this.withdraw(amount);
```

Il parametro `this` contiene un riferimento al parametro implicito, che può riferirsi a un esemplare di `BankAccount` o di una sua sottoclasse.

Il programma seguente invoca i metodi polimorfici `withdraw` e `deposit`. Dovreste prevedere manualmente cosa deve visualizzare il programma come saldo di ciascun conto, controllando che siano stati invocati veramente i metodi corretti.

### File ch10/accounts/AccountTester.java

```
/**
 * Questo programma collauda la classe BankAccount
 * e le sue sottoclassi.
 */
public class AccountTester
{
    public static void main(String[] args)
    {
        SavingsAccount momSavings = new SavingsAccount(0.5);

        CheckingAccount harrySChecking = new CheckingAccount(100);

        momSavings.deposit(10000);

        momSavings.transfer(2000, harrySChecking);
        harrySChecking.withdraw(1500);
        harrySChecking.withdraw(80);
```

```

momSavings.transfer(1000, harrysChecking);
harrysChecking.withdraw(400);

// simulazione della fine del mese
momSavings.addInterest();
harrysChecking.deductFees();

System.out.println("Mom's savings balance: "
+ momSavings.getBalance());
System.out.println("Expected: 7035");

System.out.println("Harry's checking balance: "
+ harrysChecking.getBalance());
System.out.println("Expected: 1116");
}
}

```

### Esecuzione del programma

```

Mom's savings balance: 7035.0
Expected: 7035
Harry's checking balance: 1116.0
Expected: 1116

```



### Auto-valutazione

13. Se `a` è una variabile di tipo `BankAccount` che contiene un riferimento diverso da `null`, che informazioni abbiamo in merito all'oggetto a cui si riferisce?
14. Se `a` si riferisce a un conto corrente, che effetto ha l'invocazione `a.transfer(1000, a)`?



### Argomenti avanzati 10.1

#### Classi astratte

Quando estendete una classe esistente, potete scegliere se ridefinire o meno i metodi della superclasse, ma, talvolta, conviene *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 qualche commissione mensile. Di conseguenza, aggiungiamo alla classe `BankAccount` il metodo `deductFees`:

```

public class BankAccount
{
    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 ridefinire 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*, in questo modo

```
public abstract void deductFees();
```

**Un metodo astratto è un metodo di cui non viene specificata l'implementazione.**

Un metodo astratto non ha implementazione, obbligando così i programmati di sottoclassi a specificare implementazioni concrete per questo metodo (naturalmente, per 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 `BankAccount` avesse un metodo astratto, il compilatore segnalerebbe un errore conseguente al tentativo di crearne un esemplare, con `new BankAccount()`. Naturalmente, se la sottoclasse `CheckingAccount` sovrascrive il metodo `deductFees` e ne fornisce una realizzazione, potete creare oggetti di tipo `CheckingAccount`.

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 BankAccount
{
    public abstract void deductFees();
    ...
}
```

Una classe che dichiara un metodo astratto, o che eredita un metodo astratto senza sovrascriverlo, *deve* essere dichiarata astratta. Potete dichiarare astratte anche classi prive di metodi astratti: in questo modo, evitate che si costruiscano esemplari di quella classe, ma consentite la creazione di sottoclassi.

Noteate 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:

```
BankAccount anAccount; // corretto
anAccount = new BankAccount(); // Errore: BankAccount è 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.

Le classi astratte differiscono dalle interfacce per un aspetto importante: possono avere variabili di esemplare, metodi concreti e costruttori.



## Argomenti avanzati 10.2

### Metodi e classi final

In Argomenti avanzati 10.1, avete visto come potete obbligare altri programmati a creare sottoclassi di classi astratte, sovrascrivendone i metodi astratti. Occasionalmente, potreste

desiderare il contrario e voler *impedire* ad altri programmati di creare sottoclassi o di sovrascrivere 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`.

La classe `String` è stata creata per essere *immutable*: gli oggetti stringa non possono essere modificati da nessuno dei metodi della classe. Dal momento che il linguaggio Java non lo rendeva obbligatorio, l'hanno fatto i progettisti della classe: nessuno può creare sottoclassi di `String`, per cui sapete che tutti i riferimenti a `String` si possono copiare senza correre il rischio che le stringhe vengano modificate.

Potete dichiarare `final` anche metodi singoli:

```
public class SecureAccount extends BankAccount
{
    ...
    public final boolean checkPassword(String password)
    {
        ...
    }
}
```

In questo modo, nessuno potrà sovrascrivere il metodo `checkPassword` con un altro metodo che restituisca semplicemente `true`.



## Errori comuni 10.5

### Sovrascrivere metodi rendendoli meno accessibili

Se una superclasse dichiara che un metodo ha accesso pubblico, non potete sovrascriverlo per renderlo più privato. Ecco un esempio:

```
public class BankAccount
{
    public void withdraw(double amount) { ... }
    ...

    public CheckingAccount
    {
        private void withdraw(double amount) { ... }
        // Errore: un metodo di sottoclasse non può essere più privato
        ...
    }
}
```

Il compilatore non permette questa operazione, perché la maggiore riservatezza sarebbe in conflitto con il polimorfismo. Supponete che la classe `AccountTester` contenga questo frammento di codice:

```
BankAccount account = new CheckingAccount();
account.withdraw(100000); // deve invocare CheckingAccount.withdraw ?
```

Il polimorfismo impone che venga invocato il metodo `CheckingAccount.withdraw`, ma è un metodo privato, non accessibile da `AccountTester`.

Di conseguenza, se sovrascrivete un metodo pubblico attribuendogli accesso privato o di pacchetto, il compilatore segnala un errore. In genere si tratta di una svista: se dimenticate il modificatore `public`, il vostro metodo di sottoclasse avrà l'accesso di pacchetto, che è più restrittivo. Semplicemente, ripristinate il modificatore `public`, e l'errore sparirà.



## Argomenti avanzati 10.3

### Accesso protetto

Nel tentativo di realizzare il metodo `deposit` della classe `CheckingAccount` abbiamo incontrato una serie di difficoltà, perché tale metodo aveva bisogno di accedere alla variabile di esemplare `balance` della superclasse. La nostra soluzione ha usato i metodi appropriati della superclasse per impostare il saldo.

Java offre un'altra possibilità per risolvere questo problema: la superclasse può dichiarare una variabile di esemplare *protetta*:

```
public class BankAccount
{
    ...
    protected double balance;
}
```

**Alle caratteristiche protette si può accedere da tutte le sottoclassi e da tutte le classi che si trovano nello stesso pacchetto.**

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 `CheckingAccount` eredita da `BankAccount`, quindi i suoi metodi possono accedere alle variabili di esemplare `protected` della classe `BankAccount`. In aggiunta, si può accedere ai dati protetti da tutti i metodi di classi che si trovano nello stesso pacchetto.

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 8.5).

È meglio lasciare tutti i dati privati. Se volete concedere l'accesso ai dati solo ai metodi di sottoclasse, considerate la possibilità di dichiarare protetto un metodo *di ispezione*.



## Consigli pratici 10.1

### Progettare una gerarchia di ereditarietà

Quando si lavora con un insieme di classi, alcune delle quali più generiche e altre più specifiche, le si vuole organizzare in una gerarchia di ereditarietà, per poter elaborare oggetti di classi diverse in modo omogeneo.

Per illustrare il processo di progettazione, consideriamo un'applicazione che presenta all'utente un questionario e ne valuta le risposte. Un questionario è composto da domande di tipi diversi:

- Con spazi da riempire
- A scelta (singola o multipla) tra risposte predefinite
- Numeriche (nelle quali è accettabile una risposta approssimata: 1.33 è la risposta corretta per il risultato dell'operazione  $4/3$ )
- Aperta

**Fase 1** Elencate le classi che fanno parte della gerarchia

Analizzando la descrizione del problema, identifichiamo queste classi:

`FillInQuestion` (per le domande con spazi da riempire)

`ChoiceQuestion` (presenta risposte all'utente, che dovrà scegliere)

`MultiChoiceQuestion` (presenta risposte all'utente, che potrà sceglierne più di una)

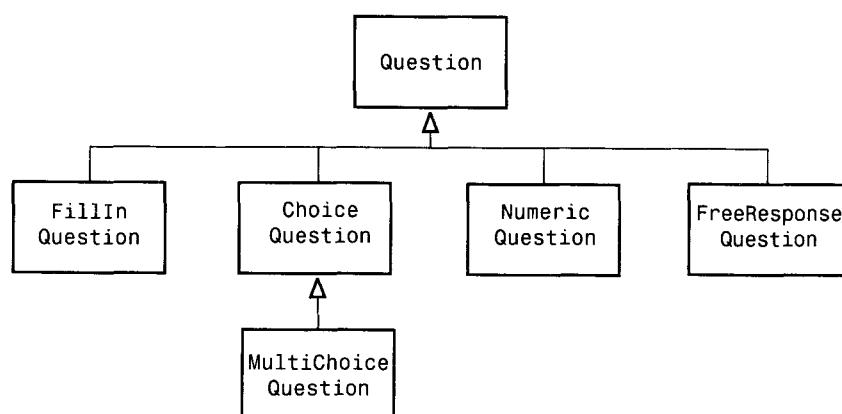
`NumericQuestion` (per le domande con risposta numerica)

`FreeResponseQuestion` (per le domande con risposta aperta)

Introduciamo, inoltre, la superclasse comune, `Question`, che rappresenta le funzionalità comuni a tutte queste classi.

**Fase 2** Organizzate le classi in una gerarchia di ereditarietà

Disegnate un diagramma UML che mostri superclassi e sottoclassi. Ecco il diagramma relativo al nostro esempio.



**Fase 3** Determinate i compiti comuni

Nella Fase 2 avrete identificato una classe come radice della gerarchia: dovete decidere quali compiti affidarle e, per farlo, è bene scrivere lo pseudocodice che serve a elaborare le domande.

Per ogni domanda

Visualizza la domanda all'utente

Acquisisci la risposta dell'utente

Verifica se la risposta è corretta

Analizzando lo pseudocodice, otteniamo questo elenco di attività che devono essere svolte dalle domande di qualsiasi tipo:

Visualizza la domanda

Verifica la risposta

**Fase 4** Decidete quali metodi vanno sovrascritti nelle sottoclassi

Per ogni sottoclasse e per ogni attività precedentemente identificata come comune, decide se il comportamento ereditato è adeguato o se c'è bisogno di ridefinirlo. Assicuratevi di dichiarare nella radice della gerarchia tutti i metodi che vanno ereditati o sovrascritti.

Inseriamo nella superclasse `Question` le funzionalità comuni a tutti i tipi di domande.

```
public class Question
{
    ...
    /**
     * Visualizza questa domanda.
     */
    public void display() { ... }

    /**
     * Verifica la correttezza di una risposta ricevuta.
     * @param response la risposta da verificare
     * @return true se la risposta è corretta, false altrimenti
     */
    public boolean checkAnswer(String response) { ... }
}
```

La classe `ChoiceQuestion` dovrà sovrascrivere il metodo `display` per fare in modo che visualizzi tutte le alternative disponibili. La classe `NumericQuestion` dovrà sovrascrivere il metodo `checkAnswer`, convertendo la risposta in un numero e verificando che sia approssimativamente uguale alla risposta prevista.

Da qui in avanti, esamineremo in dettaglio soltanto la classe `ChoiceQuestion`; per gli altri tipi di domande, fate riferimento agli esercizi che si trovano al termine del capitolo.

**Fase 5** Definite l'interfaccia pubblica di ciascuna sottoclasse

Solitamente le sottoclassi hanno funzionalità e responsabilità aggiuntive rispetto a quelle della superclasse: elencatele, insieme ai metodi che occorre sovrascrivere. Dovete altresì specificare come vadano costruiti gli oggetti delle sottoclassi.

Per quanto riguarda la classe `ChoiceQuestion`, abbiamo bisogno di un modo per aggiungere possibili risposte, ad esempio così:

```
ChoiceQuestion question = new ChoiceQuestion(
    "In which country was the inventor of Java born?");
question.addChoice("Australia", false);
question.addChoice("Canada", true);
question.addChoice("Denmark", false);
question.addChoice("United States", false);
```

Poi, sovrascriviamo il metodo `display`, in modo che visualizzi le risposte disponibili in questo formato:

```
1: Australia
2: Canada
3: Denmark
4: United States
```

Ecco, quindi, i metodi che abbiamo individuato come adeguati per la classe `ChoiceQuestion`:

```
public class ChoiceQuestion extends Question
{
    ...
    /**
     * Aggiunge una risposta tra cui scegliere per questa domanda.
     * @param choice la risposta da aggiungere
     * @param correct true se questa è la risposta corretta, false altrimenti
    */
    public void addChoice(String choice, boolean correct) { ... }

    public void display() { ... } // sovrascrive il metodo della superclasse
}
```

#### Fase 6 Individuate le variabili di esemplare

Elencate le variabili di esemplare di ciascuna classe. Se trovate una variabile di esemplare comune a tutte le classi, assicuratevi di trasferirla nella base della gerarchia.

Tutte le domande hanno un testo e una risposta: memorizziamo, quindi, questi valori nella superclasse `Question`.

```
public class Question
{
    private String text;
    private String answer;
    ...
}
```

La classe `ChoiceQuestion` ha bisogno di memorizzare l'elenco di risposte disponibili per la scelta.

```
public class ChoiceQuestion extends Question
{
```

```
    private ArrayList<String> choices;
    ...
}
```

Infine, dobbiamo pensare un po' a come vengono costruiti gli oggetti che rappresentano le domande. Nel costruttore possiamo fornire il testo della domanda, ma la scelta giusta per una domanda di tipo `ChoiceQuestion` è nota soltanto nel momento in cui l'alternativa corretta viene aggiunta, per cui ci serve un metodo per impostare tale valore:

```
public class Question
{
    ...
    /**
     * Costruisce una domanda con testo assegnato e risposta corretta vuota.
     * @param questionText il testo di questa domanda
     */
    public Question(String questionText) { ... }

    /**
     * Imposta la risposta corretta per questa domanda.
     * @param correctResponse la risposta
     */
    public void setAnswer(String correctResponse) { ... }
}
```

#### Fase 7 Realizzate costruttori e metodi

I metodi della classe `Question` sono estremamente semplici:

```
public class Question
{
    ...
    public Question(String questionText)
    {
        text = questionText;
        answer = "";
    }

    public void setAnswer(String correctResponse)
    {
        answer = correctResponse;
    }

    public boolean checkAnswer(String response)
    {
        return response.equals(answer);
    }

    public void display()
    {
        System.out.println(text);
    }
}
```

Il costruttore della classe `ChoiceQuestion` deve invocare il costruttore della superclasse, per impostare il testo della domanda:

```
public ChoiceQuestion(String questionText)
{
    super(questionText);
    choices = new ArrayList<String>();
}
```

Il metodo `addChoice` deve anche impostare la risposta corretta, quando questa viene aggiunta come alternativa possibile:

```
public void addChoice(String choice, boolean correct)
{
    choices.add(choice);
    if (correct)
    {
        // converte choices.size() in stringa
        String choiceString = "" + choices.size();
        setAnswer(choiceString);
    }
}
```

Infine, il metodo `display` della classe `ChoiceQuestion` visualizza il testo della domanda, seguito dalle risposte alternative tra cui scegliere. Notate l'invocazione del metodo della superclasse.

```
public void display()
{
    super.display();
    for (int i = 0; i < choices.size(); i++)
    {
        int choiceNumber = i + 1;
        System.out.println(choiceNumber + ": " + choices.get(i));
    }
}
```

#### Fase 8 Costruite oggetti di classi diverse ed elaborateli

In questo programma esemplificativo costruiamo due domande e le presentiamo all'utente.

```
import java.util.Scanner;

public class QuestionDemo
{
    public static void main(String[] args)
    {
        Question[] quiz = new Question[2];

        quiz[0] = new Question("Who was the inventor of Java?");
        quiz[0].setAnswer("James Gosling");

        ChoiceQuestion question = new ChoiceQuestion(
            "In which country was the inventor of Java born?");
```

```
question.addChoice("Australia", false);
question.addChoice("Canada", true);
question.addChoice("Denmark", false);
question.addChoice("United States", false);
quiz[1] = question;

Scanner in = new Scanner(System.in);
for (Question q : quiz)
{
    q.display();
    System.out.print("Your answer: ");
    String response = in.nextLine();
    System.out.println(q.checkAnswer(response));
}
}
```

### Esecuzione del programma

```
Who was the inventor of Java?
Your answer: James Gosling
true
In which country was the inventor of Java born?
1: Australia
2: Canada
3: Denmark
4: United States
Your answer: 4
false
```

Potete trovare il programma completo nella cartella `ch10/questions` del pacchetto di file scaricabili per questo libro.

## Esempi completi 10.1

### Progettare una gerarchia di dipendenti per l'elaborazione delle buste paga

Dovete realizzare un programma che elabori le buste paga per diversi tipi di dipendenti.

- Un dipendente a paga oraria (*hourly employee*) viene pagato in base a un compenso orario, ma se lavora più di 40 ore a settimana, le ore in eccesso vengono pagate "una volta e mezzo".
- Un dipendente salariato (*salaried employee*) viene pagato con tariffa settimanale ("salario"), indipendentemente da quante ore ha lavorato.
- I dirigenti (*manager*) sono dipendenti salariati ai quali viene corrisposto un salario e un "bonus".

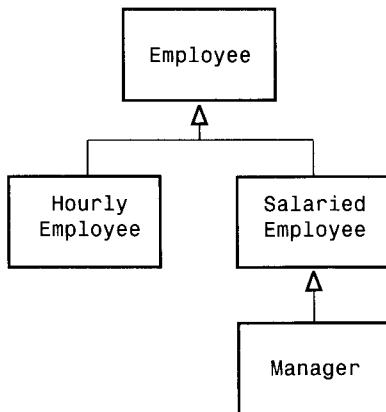
Il vostro programma ha il compito di calcolare quanto dovuto a un insieme di dipendenti. Per ciascuno di essi, chiede il numero di ore lavorate in una determinata settimana, quindi visualizza l'importo calcolato.

**Fase 1** Elencate le classi che fanno parte della gerarchia

In questo caso, la descrizione del problema elenca tre classi: `HourlyEmployee`, `SalariedEmployee` e `Manager`. Ci serve, poi, una classe che rappresenti le funzionalità comuni a tutti i dipendenti: `Employee`.

**Fase 2** Organizzate le classi in una gerarchia di ereditarietà

Ecco il diagramma UML per le nostre classi.



**Fase 3** Determinate i compiti comuni

Per identificare i compiti comuni, scriviamo lo pseudocodice che serve a elaborare i diversi oggetti.

- Per ogni dipendente
  - Visualizza il nome del dipendente
  - Leggi il numero di ore lavorate
  - Calcola quanto dovuto per quelle ore

Dall'analisi, deduciamo che la superclasse `Employee` si deve occupare di queste cose:

- Consentire l'ispezione del nome
- Calcolare quanto dovuto per un determinato numero di ore

**Fase 4** Decidete quali metodi vanno sovrascritti nelle sottoclassi

In questo esempio, per l'ispezione del nome del dipendente non c'è alcuna differenza tra i vari tipi di dipendente, ma l'importo dovuto viene calcolato in modo diverso in ciascuna sottoclasse, per cui il metodo `weeklyPay` verrà sovrascritto in tutte.

```

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 le ore lavorate
 */
public double weeklyPay(int hoursWorked) { ... }
}
```

**Fase 5** Definite l'interfaccia pubblica di ciascuna sottoclasse

Costruiamo dipendenti fornendo il nome e le informazioni relative al pagamento.

```
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) { ... }
        }
    }
}
```

Tutti questi costruttori devono impostare il nome del dipendente, per cui aggiungiamo alla classe `Employee` il metodo `setName`.

```
public class Employee
{
    ...
    public void setName(String employeeName) { ... }
    ...
}
```

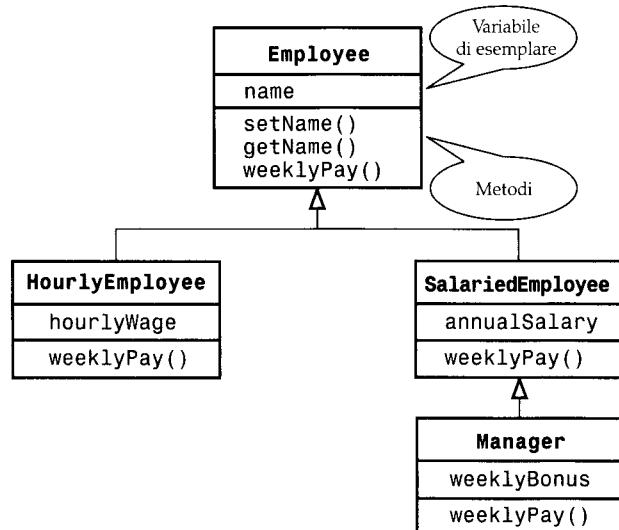
Poi, ovviamente, ogni sottoclasse deve avere un metodo che calcola l'importo dovuto per la settimana:

```
// questo metodo sovrascrive quello della superclasse
public double weeklyPay(int hoursWorked) { ... }
```

In questo semplice esempio non ci servono altri metodi.

#### Fase 6 Individuate 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 nel nuovo albero gerarchico).



Cosa possiamo dire in merito ai salari? I dipendenti a paga oraria hanno un compenso orario, mentre i salariati hanno un salario annuo. Anche se si potrebbe memorizzare questi valori in una variabile di esemplare della superclasse, non sarebbe una buona idea: il codice conseguente, che dovrebbe tener traccia dei diversi significati del numero memorizzato in tale variabile, sarebbe complesso e fonte di errori.

Decidiamo, invece, che gli oggetti di tipo `HourlyEmployee` memorizzeranno la propria paga oraria, mentre gli oggetti di tipo `SalariedEmployee` memorizzeranno il proprio salario annuo; gli oggetti di tipo `Manager` devono memorizzare anche il bonus settimanale.

#### Fase 7 Realizzate costruttori e metodi

Nelle sottoclassi, ciascun costruttore assegna un valore alle variabili di esemplare della superclasse.

```
public SalariedEmployee(String name, double salary)
{
    setName(name);
    annualSalary = salary;
}
```

Qui usiamo un metodo, mentre nel Paragrafo 10.4 avete visto come si possa invocare un costruttore della superclasse, tecnica che utilizziamo nel costruttore di Manager.

```
public Manager(String name, double salary, double bonus)
{
    super(name, salary);
    weeklyBonus = bonus;
}
```

L'importo dovuto per la settimana va calcolato seguendo le specifiche assegnate 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 ed elaboratevi

In questo programma esemplificativo popoliamo un array di dipendenti e calcoliamo i relativi importi dovuti per la settimana.

```
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));
}

```

Potete trovare il programma completo nella cartella ch10/employees del pacchetto di file scaricabili per questo libro.

## 10.7 La superclasse universale Object

Tutte le classi estendono, direttamente o indirettamente, la classe `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 7).

I metodi della classe `Object` sono, ovviamente, assai generici. Ecco quelli più utili, che, come vedremo in questo paragrafo, sarebbe sempre opportuno sovrascrivere:

| Metodo                                          | Comportamento                                  |
|-------------------------------------------------|------------------------------------------------|
| <code>String toString()</code>                  | Restituisce una stringa che descrive l'oggetto |
| <code>boolean equals(Object otherObject)</code> | Verifica se l'oggetto è uguale a un altro      |
| <code>Object clone()</code>                     | Crea una copia completa dell'oggetto           |

### 10.7.1 Sovrascrivere il metodo `toString`

Definite il metodo `toString` in modo che descriva lo stato dell'oggetto.

Il metodo `toString` restituisce una rappresentazione in forma di stringa per ciascun oggetto ed è quindi utile, ad esempio, per scovare errori logici. Ecco un esempio:

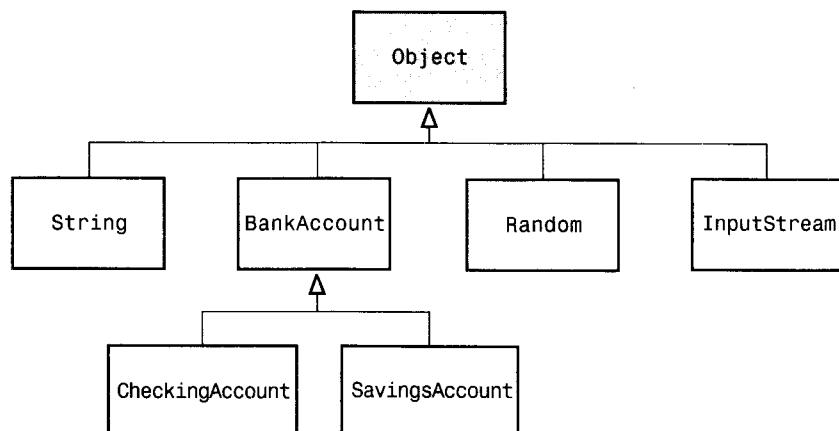
```

Rectangle box = new Rectangle(5, 10, 20, 30);
String s = box.toString();
// "java.awt.Rectangle[x=5,y=10,width=20,height=30]"

```

**Figura 7**

La classe `Object` è la superclasse di tutte le classi Java



Di fatto, questo metodo `toString` viene invocato 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 oggetto. Il compilatore Java invoca automaticamente il metodo `toString` per convertire 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, perfino i numeri vengono convertiti in stringhe quando si concatenano con altre stringhe. Per esempio:

```
int age = 18;
String s = "Harry's age is " + age;
// s diventa uguale a "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 numerici 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"
```

Il risultato è deludente: ciò che viene stampato è il nome della classe seguito dal codice hash 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 nel Capitolo 15, è molto probabile che oggetti diversi abbiano un diverso codice hash.

A noi, però, non interessa il codice hash: 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 + "]";
    }
}
```

Ora funziona meglio:

```
BankAccount momSavings = new BankAccount(5000);
String s = momSavings.toString();
// s diventa uguale a "BankAccount{balance=5000}"
```

### 10.7.2 Sovrascrivere il metodo equals

Definite il metodo `equals` in modo che verifichi se due oggetti hanno identiche informazioni di stato.

```
if (coin1.equals(coin2))
    // hanno identico contenuto, osservate la Figura 8
```

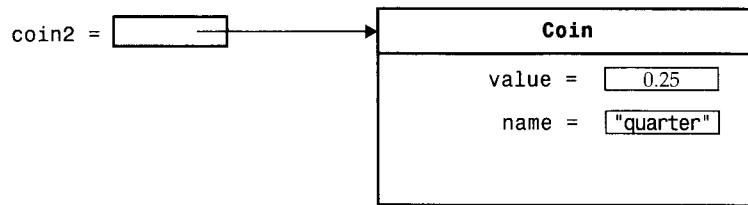
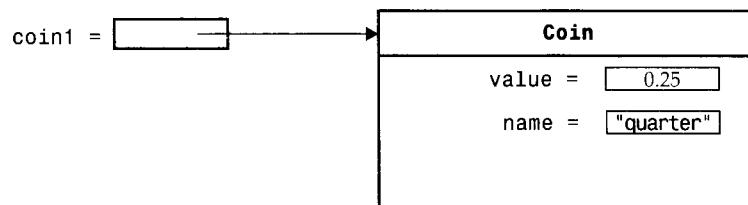
Esiste una differenza rispetto al confronto mediante l'operatore `==`, che verifica, invece, se due riferimenti puntano al *medesimo* oggetto:

```
if (coin1 == coin2)
    // sono riferimenti al medesimo oggetto, osservate la Figura 9
```

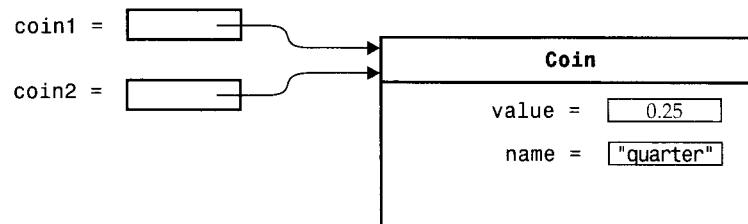
Realizziamo il metodo `equals` nella classe `Coin`, sovrascrivendo il metodo `equals` della classe `Object`:

```
public class Coin
{
    ...
}
```

**Figura 8**  
Riferimenti a oggetti uguali



**Figura 9**  
Riferimenti al medesimo oggetto



```

public boolean equals(Object otherObject)
{
    ...
}
...
}

```

Ora abbiamo un piccolo problema: la classe `Object` non sa nulla di monete, quindi dichiara il parametro `otherObject` del metodo `equals` in modo che sia di tipo `Object`. Quando sovrascrivete il metodo, non potete modificare il tipo del parametro. Per risolvere questo problema, eseguite un cast sul parametro, per convertirlo nel tipo `Coin`:

```
Coin other = (Coin) otherObject;
```

Ora potete confrontare le due monete.

```

public boolean equals(Object otherObject)
{
    Coin other = (Coin) otherObject;
    return name.equals(other.name) && value == other.value;
}

```

Notate che dovete usare `equals` per confrontare riferimenti, mentre usate `==` per confrontare numeri.

Se sovrascrivete il metodo `equals`, dovreste sovrascrivere anche il metodo `hashCode`, in modo che oggetti uguali abbiano lo stesso codice hash, come vedrete in dettaglio nel Capitolo 15.

### 10.7.3 Il metodo `clone`

Sapete già che copiando il riferimento a un oggetto si ottengono semplicemente due riferimenti al medesimo oggetto:

```

BankAccount account = new BankAccount(1000);
BankAccount account2 = account;
account2.deposit(500);
// ora sia account sia account2 puntano a un conto avente saldo 1500

```

Il metodo `clone` crea un nuovo oggetto avente lo stesso stato di un oggetto esistente.

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 10).

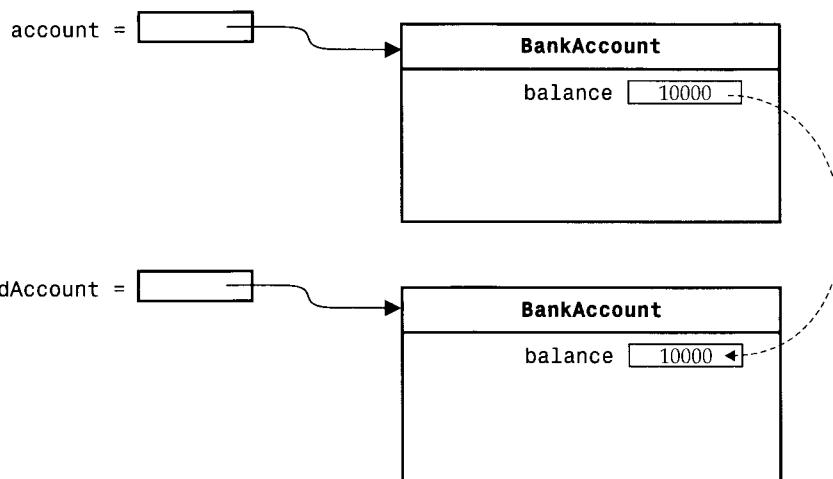
La realizzazione del metodo `clone` è un po' più difficile della realizzazione dei metodi `toString` e `equals`: consultate Argomenti avanzati 10.6 per maggiori dettagli.

Supponiamo che qualcuno abbia realizzato il metodo `clone` per la classe `BankAccount`. Ecco come invocarlo:

```
BankAccount clonedAccount = (BankAccount) account.clone();
```

Il tipo di dato che viene restituito dal metodo `clone` è `Object`, quindi, quando invocate il metodo, dovete usare un cast per convincere il compilatore che il riferimento restituito da `account.clone()` è veramente dello stesso tipo di `clonedAccount`.

**Figura 10**  
Un oggetto clonato



## Auto-valutazione

15. L'invocazione `x.equals(x)` deve sempre restituire `true`?
16. Si può realizzare il metodo `equals` usando il metodo `toString`? È una cosa consigliabile?



## Consigli per la qualità 10.1

### Realizzate il metodo `toString` in tutte le classi

Se avete una classe il cui metodo `toString()` restituisce una stringa che descrive lo stato dell'oggetto, ogni volta che dovete controllare lo stato corrente di un oggetto `x` potete scrivere semplicemente `System.out.println(x)`. Questa strategia funziona perché, quando deve visualizzare un oggetto, il metodo `println` della classe `PrintStream` invoca `x.toString()`. Tutto ciò è estremamente utile se, a causa di un errore nel programma, i vostri oggetti non si comportano nel modo previsto: potete inserire semplicemente alcuni enunciati di stampa e osservare lo stato dell'oggetto durante l'esecuzione. Alcuni programmi per il collaudo possono perfino invocare il metodo `toString` sugli oggetti che state esaminando.

Sicuramente è un po' fastidioso scrivere un metodo `toString` quando non siete certi che il vostro programma ne abbia bisogno: dopotutto, potrebbe funzionare correttamente al primo tentativo. Ma, d'altra parte, molti programmi non funzionano subito: pertanto, appena scoprite che il vostro programma non va, valutate la possibilità di inserire questi metodi `toString`, che vi saranno di aiuto.



## Argomenti avanzati 10.4

### L'ereditarietà e il metodo `toString`

Avete appena visto come scrivere un metodo `toString`: componete una stringa formata dal nome della classe e dalle coppie nome=valore delle variabili di esemplare. 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 il nome della classe nel metodo, dovreste invocare il metodo `getClass` per ottenere un oggetto di tipo `classe`, cioè un esemplare della classe `Class`, che descrive le classi e le loro proprietà. Quindi, invocate 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, anche nella sottoclasse `SavingsAccount` dovreste sovrascrivere `toString`, 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 può accedere direttamente.

```
public class SavingsAccount extends BankAccount
{
    public String toString()
    {
        return super.toString() + "[interestRate=" + interestRate + "]";
    }
}
```

Ora, un conto di risparmio viene convertito in una stringa del tipo `SavingsAccount[balance=10000][interestRate=5]`. Le parentesi quadre indicano quali variabili di esemplare appartengono alla superclasse.

## Errori comuni 10.6

### Sovrascrivere il metodo `equals` con un tipo di parametro errato

Considerate la seguente versione, apparentemente più semplice, del metodo `equals` per la classe `Coin`:

```
public boolean equals(Coin other) // NON FATELO!
{
    return name.equals(other.name) && value == other.value;
}
```

Il parametro del metodo `equals` è di tipo `Coin`, non più di tipo `Object`.

Sfortunatamente, questo metodo *non sovrscrive* il metodo `equals` della classe `Object`. La classe `Coin` viene, invece, ad avere due diversi metodi `equals`:

```
boolean equals(Coin other) // dichiarato nella classe Coin
boolean equals(Object otherObject) // ereditato dalla classe Object
```

Questa situazione è una possibile fonte di errori, perché può essere invocato il metodo `equals` sbagliato. Considerate, ad esempio, queste dichiarazioni di variabili:

```
Coin aCoin = new Coin(0.25, "quarter");
Object anObject = new Coin(0.25, "quarter");
```

L'invocazione `aCoin.equals(anObject)` chiama in causa il secondo metodo `equals`, che restituisce `false`.

La soluzione consiste nell'assicurarsi di usare il tipo `Object` come parametro esplicito nel metodo `equals`.



## Argomenti avanzati 10.5

### L'ereditarietà e il metodo `equals`

Avete appena visto come scrivere un metodo `equals`: eseguite un cast sul parametro `otherObject`, in modo da trasformarlo nel tipo della vostra classe, quindi confrontate le variabili di esemplare del parametro implicito e dell'altro parametro.

Ma cosa succede se qualcuno invoca `coin1.equals(x)`, ma `x` non è un oggetto di tipo `Coin`? Il cast errato genera un'eccezione e il programma termina bruscamente: bisogna per prima cosa verificare che `otherObject` sia realmente un esemplare della classe `Coin`. 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 `Coin`. 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;

    Coin other = (Coin) otherObject;
    return name.equals(other.name) && value == other.value;
}
```

Quando realizzate in una sottoclasse il metodo `equals`, dovreste prima di tutto invocare il metodo `equals` della superclasse, in questo modo:

```
public CollectibleCoin extends Coin
{
    private int year;
    ...
    public boolean equals(Object otherObject)
```

```

    {
        if (!super.equals(otherObject)) return false;

        CollectibleCoin other = (CollectibleCoin) otherObject;
        return year == other.year;
    }
}

```

## Consigli per la qualità 10.2

**Nei metodi d'accesso, clonate le variabili di esemplare modificabili**

Osservate questa classe:

```

public class Customer
{
    private String name;
    private BankAccount account;

    public Customer(String aName)
    {
        name = aName;
        account = new BankAccount();
    }

    public String getName()
    {
        return name;
    }

    public BankAccount getAccount();
    {
        return account;
    }
}

```

Sembra banale e comune, ma il metodo `getAccount` ha una proprietà curiosa, ovvero *viola l'incapsulamento*, dal momento che chiunque può modificare lo stato dell'oggetto senza passare attraverso l'interfaccia pubblica:

```

Customer harry = new Customer("Harry Handsome");
BankAccount account = harry.getAccount();
    // chiunque può prelevare denaro!
account.withdraw(100000);

```

Forse questo non era esattamente il risultato che i progettisti della classe avevano in mente? Forse volevano solo che gli utenti della classe potessero controllare il saldo? In un caso del genere, dovreste *clonare* l'oggetto, anziché restituirne il riferimento:

```

public BankAccount getAccount();
{
    return (BankAccount) account.clone();
}

```



Dobbiamo clonare anche nel metodo `getName()`? No, perché restituisce una stringa e le stringhe sono oggetti immutabili: non si corrono rischi se si restituisce un riferimento a un oggetto immutabile.



## Argomenti avanzati 10.6

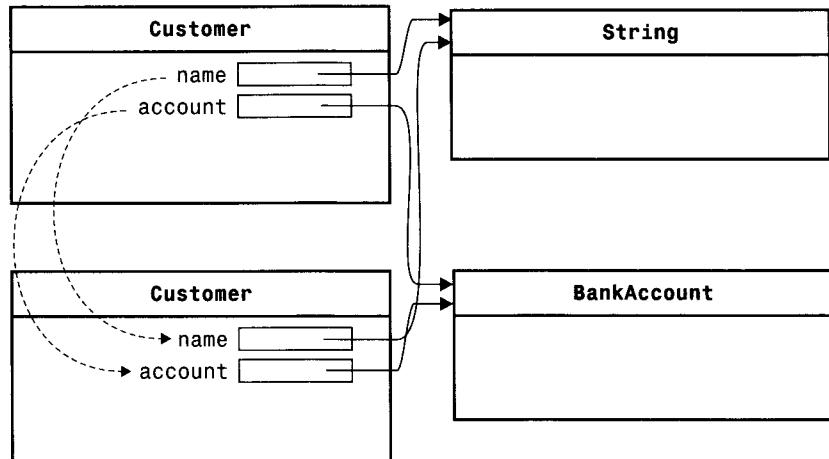
### Realizzare il metodo `clone()`

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 le variabili di esemplare dall'oggetto originario all'oggetto clonato. 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;
    }
}
```

Tuttavia, bisogna usare questo metodo 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 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").

Il metodo `Object.clone()`  
fa una copia superficiale



C'è un motivo che induce il metodo `Object.clone` a non clonare sistematicamente tutti i sotto-oggetti: in alcune situazioni, ciò 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 ad altri oggetti.

Per questo motivo esistono due protezioni, predisposte nel metodo `Object.clone`, per garantire che esso non venga utilizzato accidentalmente. Per prima cosa, il metodo è dichiarato `protected` (consultate Argomenti avanzati 10.3): questo vi impedisce di invocare `x.clone()` per sbaglio, se la classe a cui appartiene `x` non ha ridefinito `clone` come metodo pubblico.

Quale seconda precauzione, `Object.clone` controlla che l'oggetto da clonare realizzi 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 realizza `Cloneable`.

```
public class BankAccount implements Cloneable
{
    ...
    public Object clone()
    {
        try
        {
            return super.clone();
        }
        catch (CloneNotSupportedException e)
        {
            // non può verificarsi perché realizziamo 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 verificarsi perché realizziamo Cloneable
            return null;
        }
    }
}
```

## Argomenti avanzati 10.7

### Una rivisitazione dei tipi enumerativi

In Argomenti avanzati 5.3 abbiamo presentato il concetto di tipo enumerativo: un tipo con un insieme finito di valori. Ecco un esempio:

```
public enum FilingStatus { SINGLE, MARRIED }
```

In Java, i tipi enumerativi sono classi dotate di speciali proprietà: hanno un numero finito di esemplari, che sono gli oggetti definiti all'interno delle parentesi graffe. Ad esempio, esistono esattamente due esemplari della classe `FilingStatus`: `FilingStatus.SINGLE` e `FilingStatus.MARRIED`. Dato che `FilingStatus` non ha costruttori pubblici, non è possibile costruirne ulteriori esemplari.

La classi enumerative estendono la classe `Enum`, dalla quale ereditano i metodi `toString` e `clone`. Il metodo `toString` restituisce una stringa uguale al nome dell'oggetto: ad esempio, `FilingStatus.SINGLE.toString()` restituisce "SINGLE". Il metodo `clone` restituisce l'oggetto a cui viene applicato, *senza farne una copia*, anche perché deve essere assolutamente impossibile creare nuovi esemplari di una classe enumerativa.

La classe `Enum` eredita il metodo `equals` dalla sua superclasse, `Object`, per cui due costanti enumerative vengono considerate uguali solamente quando sono identiche, cioè quando sono il medesimo oggetto.

A una classe enumerativa potete aggiungere metodi e costruttori, come in questo esempio:

```

public enum CoinType
{
    private double value;
    PENNY(0.01), NICKEL(0.05), DIME(0.1), QUARTER(0.25);
    CoinType(double aValue) { value = aValue; }
    public double getValue() { return value; }
}

```

Di questa classe esistono esattamente quattro esemplari: `CoinType.PENNY`, `CoinType.NICKEL`, `CoinType.DIME` e `CoinType.QUARTER`. Con ognuno di questi quattro oggetti di tipo `CoinType` è possibile invocare il metodo `getValue`, ottenendo il valore della moneta.

Notate che esiste un'importante differenza “filosofica” fra questa classe `CoinType` e la classe `Coin` di cui abbiamo parlato in qualche altro punto di questo capitolo. Un oggetto di tipo `Coin` rappresenta una particolare moneta: potete costruire tutte le monete che volete e monete diverse possono essere fra loro uguali, perché due esemplari di `Coin` vengono considerati uguali quando i loro nomi e i loro valori coincidono. Un'esemplare di `CoinType`, invece, descrive un *tipo* di moneta, non una singola moneta, e i quattro esemplari di `CoinType` sono tra loro distinti.

## 10.8 L'ereditarietà per personalizzare i frame

Per un frame complesso definite una sottoclasse di `JFrame`.

Aggiungendovi molti componenti dell’interfaccia utente, un frame può diventare abbastanza complesso: per frame che contengono molti componenti dovreste usare l’ereditarietà, che agevola la comprensione del programma risultante.

Progettate una sottoclasse di `JFrame` e memorizzatene i componenti in variabili di esemplare, inizializzandole nel costruttore della vostra sottoclasse; se il codice del costruttore diventa troppo complesso, aggiungete semplicemente metodi ausiliari.

Seguiamo ora questo procedimento per realizzare nuovamente il programma che visualizza investimenti, già visto nel Capitolo 9.

```

public class InvestmentFrame extends JFrame
{
    private JButton button;
    private JLabel label;
    private JPanel panel;
    private BankAccount account;

    public InvestmentFrame()
    {
        account = new BankAccount(INITIAL_BALANCE);

        // usiamo variabili di esemplare per i componenti
        label = new JLabel("balance: " + account.getBalance());

        // usiamo metodi ausiliari
        createButton();
        createPanel();

        setSize(FRAME_WIDTH, FRAME_HEIGHT);
    }
}

```



## Note di cronaca 10.1

### Linguaggi di scripting

Immaginate di lavorare in un ufficio dove avete a che fare con scritture contabili. Immaginate, ancora, che ciascun responsabile delle vendite invii settimanalmente un foglio elettronico con i dati di vendita e che uno dei vostri compiti sia quello di trascrivere tali dati singoli in un foglio elettronico riassuntivo, per poi copiarne i totali in un documento di testo che venga inviato tramite la posta elettronica a diversi dirigenti. Questo tipo di lavoro può essere molto noioso: potete pensare di automatizzarlo?

Sarebbe una vera sfida scrivere un programma Java che vi possa aiutare: dovreste sapere come leggere un foglio elettronico memorizzato in un file, come impaginare un documento per un elaboratore di testi e come inviare un messaggio di posta elettronica.

Fortunatamente, molti pacchetti di utilità per ufficio contengono, oggi, *linguaggi di scripting*: linguaggi di programmazione integrati in altri programmi allo scopo di automatizzare compiti ripetitivi. Il più noto è Visual Basic Script, che fa parte del pacchetto Microsoft Office, mentre il sistema operativo Macintosh ha un linguaggio chiamato AppleScript per svolgere funzioni simili.

Esistono linguaggi di script in molti altri ambiti: ad esempio, JavaScript è usato nelle pagine Web (non c'è alcuna relazione tra Java e JavaScript, il nome JavaScript è stato scelto per motivi commerciali); Tcl, abbreviazione per *tool control language*

e pronunciato come "tickle" (*soltellato*), è un linguaggio di scripting *open source* (di cui è, quindi, disponibile anche il codice sorgente) che funziona su molte piattaforme e che viene spesso usato per realizzare procedure di collaudo. Gli script di *shell* vengono utilizzati per automatizzare la configurazione del software, le procedure di backup e altri compiti di gestione del sistema.

I linguaggi di scripting hanno due caratteristiche che li rendono più semplici da usare rispetto a linguaggi di programmazione veri e propri come Java. Innanzitutto, sono *interpretati*: un programma che funge da interprete legge ciascuna linea di codice del programma e la esegue immediatamente, senza alcuna attività di compilazione preventiva. Ciò rende la sperimentazione

#### Uso di classi Java mediante JavaScript

ancora più divertente, perché si ha un riscontro immediato. Ancora, i linguaggi di scripting sono spesso *debolmente tipizzati*: ciò significa che non c'è bisogno di dichiarare i tipi delle variabili, perché ciascuna variabile può contenere valori di qualsiasi tipo. La figura mostra un esempio di utilizzo della realizzazione di JavaScript contenuta nel Java Development Kit.

Questa versione di JavaScript consente la manipolazione di oggetti Java. Lo script memorizza oggetti di tipo frame e oggetti di tipo etichetta in variabili che sono dichiarate senza specificare alcun tipo. I metodi invocati, poi, vengono eseguiti immediatamente, senza compilazione: la finestra frame compare sullo schermo appena viene digitata la linea che contiene il comando `setVisible`. Recentemente gli autori di virus per computer hanno scoperto come

```

Terminal
File Edit View Terminal Tabs Help
~$ jrunscript
js> importPackage(Packages(javax.swing));
js> frame = new JFrame();
javax.swing.JFrame[frame0.0.0.0x0.invalid.hidden.layout=java.awt.BorderLayout,t
itle=resizable,normal,defaultCloseOperation=HIDE_ON_CLOSE,rootPane=javax.swing.J
RootPane[0.0.0x0.invalid.layout=javax.swing.JRootPane$RootLayout,alignment=0.0
,alignmentY=0.0,border=,flags=16777673,maximumSize=,minimumSize=,preferredSize=]
,rootPaneCheckingEnabled=true]
js> label = new JLabel("Hello, World");
javax.swing.JLabel[0.0.0x0.invalid.alignmentX=0.0.alignmentY=0.0.border=,flags=
8388608,maximumSize=,minimumSize=,preferredSize=,defaultIcon=,disabledIcon=,horiz
ontalAlignment=LEADING,horizontalTextPosition=TRAILING,iconTextGap=4,labelFor=,
text>Hello, World,verticalAlignment=CENTER,verticalTextPosition=CENTER]
js> frame.add(label);
javax.swing.JFrame[frame0.0.0x0.invalid.alignmentX=0.0.alignmentY=0.0.border=,flags=
8388608,maximumSize=,minimumSize=,preferredSize=,defaultIcon=,disabledIcon=,horiz
ontalAlignment=LEADING,horizontalTextPosition=TRAILING,iconTextGap=4,labelFor=,
text>Hello, World,verticalAlignment=CENTER,verticalTextPosition=CENTER]
js> frame.setSize(200, 100);
js> frame.setVisible(true);
js> 

```

I linguaggi di scripting possano semplificare loro la vita. Ad esempio, il famoso "love bug" è un programma in Visual Basic Script che viene inserito in un messaggio di posta elettronica. Il messaggio ha un titolo illettante, "I love you", e chiede al destinatario di selezionare e attivare un allegato mascherato da lettera d'amore: in realtà, l'allegato è un file che contiene un programma script che viene eseguito quando l'utente lo seleziona. Lo script provoca dei danni nel calcolatore del destinatario

del messaggio e, sfruttando la potenza del linguaggio di scripting, usa il programma Outlook per inviare se stesso come messaggio di posta elettronica a tutti gli indirizzi che trova nella rubrica. Provate a scrivere un programma simile in Java! In ogni caso, la persona sospettata di aver scritto quel virus è uno studente che scrisse una proposta di tesi di ricerca su come scrivere programmi di quel tipo: forse non c'è da meravigliarsi che tale proposta sia stata bocciata dal collegio dei docenti.

Perché abbiamo comunque bisogno di Java, se usare i linguaggi di scripting è facile e divertente? I programmi script hanno un debole controllo sugli errori e sono difficilmente adattabili a nuove situazioni. I linguaggi di scripting mancano di molte delle strutture e dei meccanismi di sicurezza (come le classi e il controllo dei tipi svolto dal compilatore) che sono importanti per progettare programmi robusti e scalabili.

```
private void createButton()
{
    button = new JButton("Add Interest");
    ActionListener listener = new AddInterestListener();
    button.addActionListener(listener);
}

private void createPanel()
{
    panel = new JPanel();
    panel.add(button);
    panel.add(label);
    add(panel);
}
...
}
```

Questo approccio è diverso da quello visto nel Capitolo 9, dove avevamo semplicemente configurato il frame all'interno del metodo `main` di una classe di visualizzazione.

Progettare una classe separata per il frame richiede un po' di lavoro aggiuntivo, ma semplifica l'organizzazione del codice che costruisce gli elementi dell'interfaccia grafica.

Ovviamente occorre comunque una classe con il metodo `main`:

```
public class InvestmentViewer2
{
    public static void main(String[] args)
    {
        JFrame frame = new InvestmentFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```



## Auto-valutazione

17. Quanti file di codice Java sono necessari per l'applicazione che visualizza investimenti quando viene usata l'ereditarietà per definire una classe per il frame?
18. Perché il costruttore di `InvestmentFrame` invoca `setSize(FRAME_WIDTH, FRAME_HEIGHT)`, mentre il metodo `main` del programma visualizzatore di investimenti del Capitolo 9 invocava `frame.setSize(FRAME_WIDTH, FRAME_HEIGHT)`?



## Argomenti avanzati 10.8

### Aggiungere il metodo `main` alla classe del frame

Riguardate le classi `InvestmentFrame` e `InvestmentViewer2`. Alcuni programmati preferiscono combinare queste due classi, aggiungendo il metodo `main` alla classe del frame:

```
public class InvestmentFrame extends JFrame
{
    public static void main(String[] args)
    {
        JFrame frame = new InvestmentFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

    public InvestmentFrame()
    {
        account = new BankAccount(INITIAL_BALANCE);

        // usiamo variabili di esemplare per i componenti
        label = new JLabel("balance: " + account.getBalance());

        // usiamo metodi ausiliari
        createButton();
        createPanel();

        setSize(FRAME_WIDTH, FRAME_HEIGHT);
    }
    ...
}
```

Si tratta di una comoda scorciatoia che troverete in molti programmi, ma confonde i compiti della classe frame e del programma, per cui in questo libro non useremo tale approccio.

## Riepilogo degli obiettivi di apprendimento

### Le nozioni di ereditarietà, superclasse e sottoclasse

- Insiemi di classi possono dar luogo a gerarchie di ereditarietà complesse.

### Realizzare sottoclassi in Java

- L'ereditarietà è uno strumento per estendere classi esistenti aggiungendo metodi e variabili di esemplare.
- Una sottoclasse eredita i metodi della sua superclasse.
- Le variabili di esemplare dichiarate nella superclasse sono presenti anche negli oggetti di una sottoclasse.
- Una sottoclasse non ha accesso alle variabili private della sua superclasse.
- Ereditare da una classe non è come realizzare un'interfaccia: la sottoclasse eredita il comportamento della propria superclasse.

### Una sottoclasse può ridefinire metodi della sua superclasse

- Una sottoclasse può ereditare un metodo dalla superclasse oppure sovrascriverlo, fornendone una realizzazione alternativa.
- Per invocare un metodo della superclasse usate la parola riservata `super`.

### Costruzione di un esemplare di sottoclasse usando anche la superclasse

- Per invocare il costruttore della superclasse si usa la parola riservata `super` nel primo enunciato del costruttore della sottoclasse.

### Conversione di tipo tra sottoclasse e superclasse

- I riferimenti a sottoclasse possono essere convertiti in riferimenti a superclasse.
- L'operatore `instanceof` verifica se un oggetto è di un particolare tipo.

### Ricerca dinamica dei metodi e polimorfismo

- Quando la macchina virtuale invoca un metodo di esemplare, lo cerca nella classe corrispondente al parametro implicito, mediante *ricerca dinamica* del metodo.

### Ridefinizione dei metodi della superclasse `Object`

- Tutte le classi estendono, direttamente o indirettamente, la classe `Object`.
- Definite il metodo `toString` in modo che descriva lo stato dell'oggetto.
- Definite il metodo `equals` in modo che verifichi se due oggetti hanno identiche informazioni di stato.
- Il metodo `clone` crea un nuovo oggetto avente lo stesso stato di un oggetto esistente.

### Personalizzare i frame mediante ereditarietà

- Per un frame complesso definite una sottoclasse di `JFrame`.

## Classi, oggetti e metodi presentati nel capitolo

```
java.lang.Cloneable
java.lang.CloneNotSupportedException
java.lang.Object
    clone
    toString
```

## Esercizi di ripasso

- ★ **Esercizio R10.1.** Qual è il saldo di `b` dopo queste operazioni?

```
SavingsAccount b = new SavingsAccount(10);
```

```
b.deposit(5000);
b.withdraw(b.getBalance() / 2);
b.addInterest();
```

- ★ **Esercizio R10.2.** Descrivete tutti i costruttori della classe `SavingsAccount`. Elicitate tutti i metodi che sono da essa ereditati dalla classe `BankAccount`. Elicitate tutti i metodi che vengono aggiuntati alla classe `SavingsAccount`.
- ★★ **Esercizio R10.3.** Potete convertire un riferimento a una superclasse in un riferimento a una sottoclasse? È un riferimento a una sottoclasse in uno a una superclasse? In caso affermativo, fornite un esempio, altrimenti spiegatene i motivi.
- ★★ **Esercizio R10.4.** Nelle seguenti coppie di classi, individuate la superclasse e la sottoclasse.
  - a. Impiegato, Dirigente
  - b. Poligono, Triangolo
  - c. StudenteUniversitario, Studente
  - d. Persona, Studente
  - e. Impiegato, StudenteUniversitario
  - f. ContoBancario, ContoCorrenteBancario
  - g. Veicolo, Automobile
  - h. Veicolo, AutomobileFamiliare
  - i. Automobile, AutomobileFamiliare
  - j. Autocarro, Veicolo
- ★ **Esercizio R10.5.** Se la classe `Sub` estende la classe `Sandwich`, quali fra le seguenti sono assegnazioni ammesse dopo aver eseguito i primi due enunciati?
 

```
Sandwich x = new Sandwich();
Sub y = new Sub();
```

  - a. x = y;
  - b. y = x;
  - c. y = new Sandwich();
  - d. x = new Sub();
- ★ **Esercizio R10.6.** Disegnate un diagramma di ereditarietà che mostri le relazioni ereditarie fra le classi seguenti:
  - Persona
  - Impiegato
  - Studente
  - Docente
  - Aula
  - Object

- ★★ **Esercizio R10.7.** Disegnate un diagramma di ereditarietà che mostri le relazioni ereditarie fra queste classi, utilizzate in un sistema orientato agli oggetti per la simulazione di traffico:
  - Veicolo
  - Automobile
  - Autocarro
  - AutomobileBerlina
  - AutomobileSportiva
  - PiccoloAutocarro
  - AutomobileSUV

- AutomobileFamiliare
- Bicicletta
- Motociclo

\*\* **Esercizio R10.8.** Quali relazioni di ereditarietà stabilireste fra le classi seguenti?

- Studente
- Professore
- AssistenteDelProfessore
- Impiegato
- Segretaria
- DirettoreDiDipartimento
- Bidello
- OratoreDiSeminario
- Persona
- Corso
- Seminario
- Lezione
- EsercitazioneInLaboratorio

\*\*\* **Esercizio R10.9.** Quale di queste condizioni restituisce true? Controllate la documentazione di Java per identificare le relazioni di ereditarietà.

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

- a. if (r instanceof Rectangle) ...
- b. if (r instanceof Point) ...
- c. if (r instanceof Rectangle2D.Double) ...
- d. if (r instanceof RectangularShape) ...
- e. if (r instanceof Object) ...
- f. if (r instanceof Shape) ...

\*\* **Esercizio R10.10.** Spiegate i due significati della parola riservata `super`. Spiegate i due significati della parola riservata `this`. In che modo sono correlati?

\*\*\* **Esercizio R10.11.** (Insidioso) Quale di queste due invocazioni costituisce un esempio di polimorfismo?

```
public class D extends B
{
    public void f()
    {
        this.g(); // 1
    }
    public void g()
    {
        super.g(); // 2
    }
    ...
}
```

\*\*\* **Esercizio R10.12.** In questo programma, le invocazioni dei metodi `endOfMonth` vengono risolte mediante la ricerca dinamica? All'interno del metodo `printBalance`, l'invocazione di `getBalance` è risolta mediante la ricerca dinamica?

```

public class AccountPrinter
{
    public static void main(String[] args)
    {
        SavingsAccount momssSavings = new SavingsAccount(0.5);

        CheckingAccount harrysChecking = new CheckingAccount(0);

        ...
        endOfMonth(momssSavings);
        endOfMonth(harrysChecking);
        printBalance(momssSavings);
        printBalance(harrysChecking);
    }

    public static void endOfMonth(SavingsAccount savings)
    {
        savings.addInterest();
    }

    public static void endOfMonth(CheckingAccount checking)
    {
        checking.deductFees();
    }

    public static void printBalance(BankAccount account)
    {
        System.out.println("The balance is $" + account.getBalance());
    }
}

```

- ★ **Esercizio R10.13.** Spiegate i termini *copia superficiale* e *copia profonda*.
- ★ **Esercizio R10.14.** Quale attributo di accesso dovrebbero avere le variabili di esemplare? E le variabili statiche? E le costanti statiche?
- ★ **Esercizio R10.15.** Quale attributo di accesso dovrebbero avere i metodi di esemplare? La stessa regola è valida anche per i metodi statici?
- ★★ **Esercizio R10.16.** Le variabili statiche `System.in` e `System.out` sono pubbliche. È possibile modificarle? In caso affermativo, in che modo?
- ★★ **Esercizio R10.17.** Perché le variabili di esemplare pubbliche sono pericolose? Le variabili statiche pubbliche sono più pericolose delle variabili di esemplare pubbliche?

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

# 11

## Ingresso/uscita e gestione delle eccezioni

### Obiettivi del capitolo

- Essere in grado di leggere e scrivere file di testo
- Imparare a lanciare e a catturare eccezioni
- Saper progettare proprie classi di eccezioni
- Capire la differenza tra eccezioni a controllo obbligatorio ed eccezioni a controllo non obbligatorio
- Sapere quando e dove catturare un'eccezione

Questo capitolo inizia presentando la gestione di file di testo per dati in ingresso e in uscita dai programmi. Ogni volta che leggete o scrivete dei dati, ci si può attendere che si verifichino errori; inoltre, un file può essere stato corrotto nei suoi contenuti o addirittura cancellato, oppure può essere stato memorizzato in un altro calcolatore che è poi stato disconnesso dalla rete. In queste situazioni, per poter reagire, dovete conoscere la gestione delle eccezioni: questo capitolo vi spiega come i programmi possono segnalate condizioni eccezionali e come ripristinare un corretto funzionamento una volta che sia avvenuta una tale condizione anomala.

## 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 che vengono creati con un semplice editor di testi quale Windows Notepad oppure i file di codice sorgente Java o i file HTML.

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 da console. 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 file system (tipicamente archiviato in un disco); questa classe è molto articolata e 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; successivamente, si utilizza tale oggetto per costruire un oggetto di tipo `Scanner`:

```
File inFile = new File("input.txt");
Scanner in = new Scanner(inFile);
```

**Per leggere file di testo usate la classe `Scanner`.**

**Per scrivere file di testo usate la classe `PrintWriter`.**

Questo oggetto `Scanner` legge il testo contenuto nel file `input.txt` e, per leggere dati, potete usare i consueti metodi della classe (come `next`, `nextLine`, `nextInt` e `nextDouble`).

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. Potete costruire un esemplare di `PrintWriter` anche a partire da un oggetto di tipo `File`, operazione utile, ad esempio, se usate una finestra di dialogo per la selezione di file (*file chooser*, in Argomenti avanzati 11.1).

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.print(29.95);
out.println(new Rectangle(5, 10, 15, 25));
out.printf("%10.2f", price);
```

**Quando avete finito di scrivere in un file, dovete chiuderlo.**

Quando avete terminato di scrivere in un file, accertatevi di *chiudere* l’oggetto di tipo `PrintWriter`:

```
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 su disco.

Il programma seguente mette all’opera questi concetti: legge tutte le righe presenti in un file e le scrive in un altro file, inserendo all’inizio di ciascuna riga il corrispondente *numero di riga*. Se il file d’ingresso è:

```
Mary had a little lamb  
whose fleece was white as snow.  
And everywhere that Mary went,  
The lamb was sure to go!
```

allora il programma produce il seguente 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 sono racchiusi tra delimitatori `/* */` in modo che il programma possa essere utilizzato per numerare file di codice sorgente Java.

C'è, però, un ulteriore problema con cui confrontarsi: è possibile che si verifichi l'eccezione `FileNotFoundException`, ad esempio quando un file utilizzato per leggere dati non esiste, oppure quando si vuole scrivere in un file non ancora esistente che, per qualche motivo legato al sistema operativo, non può essere creato. In questo caso, il compilatore esige che gli diciamo esplicitamente come vogliamo che il nostro programma reagisca in una tale situazione (sotto questo aspetto, l'eccezione `FileNotFoundException` è diversa dalle eccezioni che abbiamo incontrato in precedenza, come vedremo in dettaglio nel Paragrafo 11.4). Nel nostro programma useremo l'approccio più semplice e dichiareremo che il metodo `main` deve semplicemente terminare la propria esecuzione nel caso in cui si verifichi l'eccezione, etichettando il metodo stesso in questo modo:

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

Nei paragrafi successivi vedremo come gestire in modo più professionale le eccezioni.

### File ch11/lines/LineNumberer.java

```
import java.io.File;  
import java.io.FileNotFoundException;  
import java.io.PrintWriter;  
import java.util.Scanner;  
  
/**  
 * Questo programma inserisce i numeri di riga in un file.  
 */  
public class LineNumberer  
{  
    public static void main(String[] args) throws FileNotFoundException  
    {  
        // chiede all'utente i nomi dei file di ingresso e di uscita  
  
        Scanner console = new Scanner(System.in);  
        System.out.print("Input file: ");  
        String inputFileName = console.next();  
        System.out.print("Output file: ");  
        String outputFileName = console.next();  
  
        // costruisce gli oggetti Scanner e PrintWriter per leggere e scrivere
```

```

File inputFile = new File(inputFileName);
Scanner in = new Scanner(inputFile);
PrintWriter out = new PrintWriter(outputFileName);
int lineNumber = 1;

// legge e scrive

while (in.hasNextLine())
{
    String line = in.nextLine();
    out.println("/* " + lineNumber + " */ " + line);
    lineNumber++;
}

in.close();
out.close();
}
}

```



## Auto-valutazione

1. Cosa succede se al programma `LineNumberer` viene fornito lo stesso nome per i file di ingresso e di uscita?
2. Cosa succede se al programma `LineNumberer` viene fornito come nome del file d'ingresso il nome di un file inesistente?



## 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 in un nome di file nei sistemi operativi Windows), dovete inserire ciascuna barra rovesciata *due volte*:

```
inFile = new File("c:\\\\homework\\\\input.dat");
```

Ricordate che una sola barra rovesciata entro stringhe racchiuse da virgolette è un *carattere di escape*, che viene combinato con un altro carattere 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 da un utente, la barra rovesciata non va digitata due volte.



## Errori comuni 11.2

### Costruire uno `Scanner` per leggere una stringa

Quando costruirete un oggetto di tipo `PrintWriter` usando una stringa come parametro di costruzione, scriverete in un file:

```
PrintWriter out = new PrintWriter("output.txt");
```

Questo, però, non funziona con un oggetto di tipo **Scanner**. L'enunciato

```
Scanner in = new Scanner("input.txt"); // ERRORE ?
```

*non* apre un file, ma, semplicemente, analizza il contenuto della stringa: l'invocazione `in.nextLine()` restituisce la stringa "input.txt". Si tratta, in alcuni casi (come nel Paragrafo 11.2.3), di una caratteristica utile.

Dovete soltanto ricordarvi di fornire al costruttore di **Scanner** un oggetto di tipo **File**:

```
Scanner in = new Scanner(new File("input.txt")); //così va bene
```

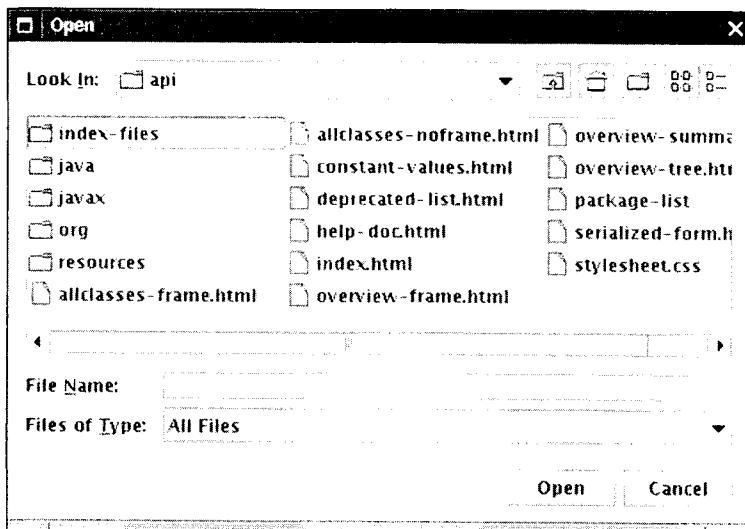


## Argomenti avanzati 11.1

### 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, come quella realizzata dalla classe **JFileChooser** nel pacchetto Swing.

Una finestra di dialogo di tipo **JFileChooser**



Una finestra di dialogo **JFileChooser** consente la selezione di un file navigando nelle cartelle.

Un oggetto di tipo **File** descrive un file o una cartella del file system.

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 oppure `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 `getSelectedFile`. 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 11.2

### Leggere pagine web

Con questa sequenza di enunciati potete leggere il contenuto di una pagina web:

```
String address = "http://java.sun.com/index.html";
URL locator = new URL(address);
Scanner in = new Scanner(locator.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 11.3 troverete maggiori informazioni sulla clausola `throws`).

## Argomenti avanzati 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*.

Per esempio, può essere comodo specificare sulla riga dei comandi i nomi del file di ingresso e di uscita per il programma `LineNumberer`:

```
java LineNumberer input.txt numbered.txt
```

**Quando eseguite un programma dalla riga dei comandi, potete specificare argomenti dopo il nome del programma stesso, che può poi accedere a quelle stringhe elaborando il parametro `args` del proprio metodo `main`.**

Le stringhe digitate dopo il nome del programma Java vengono inserite nel parametro `args` del metodo `main`: finalmente scoprirete l’utilizzo di tale parametro, che avete visto in tutti i programmi!

Ad esempio, con la riga di comando che abbiamo appena visto, il parametro `args` del metodo `LineNumberer.main` è così composto:

- `args[0]` è "input.txt"
- `args[1]` è "numbered.txt"

Una volta messo in esecuzione il programma, il metodo `main` può elaborare tali parametri, ad esempio in questo modo:

```
if (args.length >= 1)
    inputFileName = args[0];
```

Decidere come utilizzare le stringhe fornite come argomenti sulla riga dei comandi è una responsabilità che compete esclusivamente al programma, ma di solito le stringhe che iniziano con un trattino sono considerate opzioni, che modificano il funzionamento del programma. Ad esempio, possiamo migliorare il programma `LineNumberer` facendo in modo che l'opzione `-c` provochi l'inserimento dei numeri di riga all'interno di delimitatori per commenti, in questo modo:

```
java LineNumberer -c HelloWorld.java HelloWorld.txt
```

In mancanza dell'opzione `-c`, non vengono aggiunti i delimitatori. Ecco la porzione del metodo `main` che analizza gli argomenti forniti sulla riga dei comandi:

```
for (String arg : args)
{
    if (arg.startsWith("-")) // è un'opzione
    {
        if (arg.equals("-c")) useCommentDelimiters = true;
    }
    else if (inputFileName == null) inputFileName = arg;
    else if (outputFileName == null) outputFileName = arg;
}
```

Per gli utenti dei vostri programmi, è meglio un'interfaccia a riga di comandi o un'interfaccia grafica, con finestre di dialogo per la selezione dei file? Per un utente sporadico, l'interfaccia grafica è assai migliore, perché lo guida e rende possibile l'utilizzo dell'applicazione senza alcuna conoscenza, ma per un utente assiduo le interfacce grafiche hanno un grande svantaggio: sono difficili da automatizzare. Se dovete elaborare centinaia di file ogni giorno, trascorrete tutto il vostro tempo a inserire nomi di file in finestre di dialogo, mentre non è difficile, usando gli strumenti messi a disposizione dal sistema operativo (come i file batch visti nei Consigli per la produttività 7.3), invocare più volte un programma in modo automatico fornendo sulla riga di comando argomenti diversi.,

## 11.2 Acquisire testi

In questo paragrafo imparerete a elaborare in ingresso dati di tipo testo, con il livello di complessità tipico dei problemi reali.

### 11.2.1 Leggere parole

Nel programma visto precedentemente come esempio, leggevamo i dati in ingresso riga per riga, ma a volte è utile leggere parole, anziché righe. Considerate, ad esempio, questo ciclo:

```
while (in.hasNext())
{
    String input = in.next();
    System.out.println(input);
}
```

Il metodo `next` legge una parola per volta; per specificare uno schema per suddividere le parole si invoca `Scanner.useDelimiter`.

Fornendo in ingresso il testo usato nell'esempio precedente, questo ciclo visualizzerebbe una parola su ciascuna riga:

```
Mary
had
a
little
lamb
```

Il concetto di *parola*, in Java, è diverso da quello della lingua inglese: è una sequenza di caratteri che non contiene “spazi bianchi” (*white spaces*). Questi ultimi sono i caratteri di spaziatura veri e propri, i caratteri di tabulazione e i caratteri di “nuova riga” che separano le righe. Ad esempio, queste sono considerate parole:

```
snow.
1729
C++
```

(osservate che il punto dopo `snow` è considerato parte della parola, perché non è un spazio bianco).

Vediamo in dettaglio cosa accade quando viene invocato il metodo `next`. I caratteri in ingresso che sono *spazi bianchi* 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 uno spazio bianco diventa il primo carattere della parola. Si aggiungono ulteriori caratteri finché non si trova uno spazio bianco oppure si esaurisce il flusso.

A volte si vogliono leggere proprio le parole, 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 stabilito che lo schema di caratteri (“pattern”) che separa le parole è “qualunque sequenza di caratteri diversi dalle lettere”. Per descrivere lo schema abbiamo usato una notazione che viene chiamata *espressione canonica* (“regular expression”), di cui parliamo in maggiore dettaglio nei Consigli per la produttività 11.1. Cor-

queste impostazioni, i segni di punteggiatura e le cifre numeriche vengono eliminate dalle parole restituite dal metodo `next`.

### 11.2.2 Elaborare righe

Il metodo `nextLine` legge una riga in ingresso e consuma il carattere di “nuova riga” che si trova alla fine della riga letta.

```
String line = in.nextLine();
```

Il metodo `nextLine` estrae dal flusso (“consuma”) la successiva riga di testo, compreso il carattere di “nuova riga”, e restituisce la riga letta, priva del carattere terminale di “nuova riga”: si può, così, disporre della riga per le elaborazioni richieste.

Ecco un esempio tipico di elaborazione di righe di un file. Un file con dati di popolazione, estratto dal CIA Fact Book (reperibile all’indirizzo <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, dopodiché usiamo i metodi `isDigit` e `isWhitespace` 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++; }
```

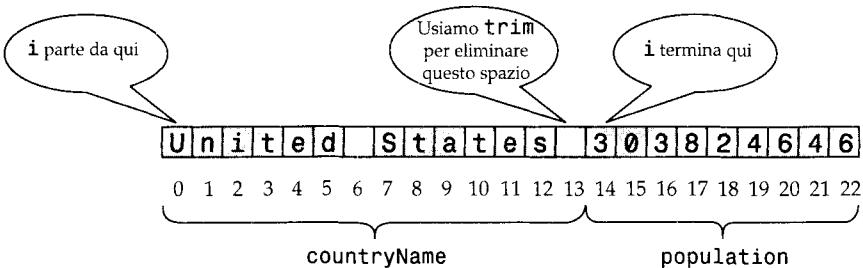
Quindi, estraiamo il nome della nazione e la relativa popolazione:

```
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 gli spazi bianchi iniziali e finali.



Abbiamo ancora un problema: la popolazione è memorizzata in una stringa, non in una variabile numerica. Usiamo il metodo `Integer.parseInt` per effettuare la conversione:

```
int populationValue = Integer.parseInt(population);
```

Quando si invoca il metodo `Integer.parseInt` occorre fare attenzione: il suo parametro deve essere una stringa che contiene le cifre di un numero intero, altrimenti viene lanciata l'eccezione `NumberFormatException`. La stringa fornita come parametro non deve contenere alcun carattere estraneo, nemmeno spazi bianchi! Nel nostro caso, 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());
```

Abbiamo visto come scomporre una stringa in componenti osservando singoli caratteri, ma a volte è più semplice usare un approccio diverso, quando è possibile, costruendo un esemplare di `Scanner` che legga caratteri dalla stringa stessa:

```
Scanner lineScanner = new Scanner(line);
```

Si può, poi, usare `lineScanner` come qualsiasi altro oggetto di tipo `Scanner`, leggendo parole e numeri:

```
String countryName = lineScanner.next();
while (!lineScanner.hasNextInt())
{
    countryName = countryName + " " + lineScanner.next();
}
int populationValue = lineScanner.nextInt();
```

### 11.2.3 Leggere numeri

I metodi `nextInt` e `nextDouble` consumano eventuali spazi bianchi e leggono il numero che segue.

Avete già usato molte volte i metodi `nextInt` e `nextDouble` della classe `Scanner`, ma qui ci occuperemo del loro comportamento in maggiore dettaglio. Con l'invocazione

```
double value = in.nextDouble();
```

il metodo `nextDouble` riconosce numeri in virgola mobile, come `3.14159`, `-21` o `1E12` (un milione di miliardi, o *bilione*, in notazione scientifica). Se, però, nei dati analizzati *non c'è alcun numero*, viene lanciata l'eccezione `NoSuchElementException`.

Ora ipotizziamo che in ingresso siano presenti questi caratteri

|   |   |   |   |   |   |   |   |   |   |   |   |  |
|---|---|---|---|---|---|---|---|---|---|---|---|--|
| 1 | 2 | 1 | s | t | c | e | n | t | u | r | y |  |
|---|---|---|---|---|---|---|---|---|---|---|---|--|

Invocando `nextDouble` viene “consumato” lo spazio iniziale e viene letta la parola `21st`, che non rispetta il formato previsto per i numeri: in questa situazione si ha l’eccezione `InputMismatchException`, per “mancata corrispondenza” (*mismatch*) tra il formato previsto e il formato effettivamente letto.

Per evitare il lancio di eccezioni, si analizza il flusso di dati in ingresso con il metodo `hasNextDouble`, ad esempio in questo modo:

```
if (in.hasNextDouble())
{
    double value = in.nextDouble();
    ...
}
```

Analogamente, prima di invocare `nextInt` bisognerebbe invocare `hasNextInt`.

Osservate che i metodi `nextInt` e `nextDouble` *non* consumano gli spazi bianchi che seguono il numero letto: ciò può rappresentare un problema se si alternano le invocazioni di `nextInt/nextDouble` e di `nextLine`. Immaginate, infatti, di avere un file contenente numeri identificativi di studenti e i loro nomi, in questo formato:

```
1729
Harry Morgan
1730
Diana Lin
...
```

e di leggerlo con questo frammento di codice:

```
while (in.hasNextInt())
{
    int studentID = in.nextInt();
    String name = in.nextLine();
    Elabora i dati dello studente
}
```

Inizialmente in ingresso abbiamo

|   |   |   |   |    |   |   |   |   |   |  |
|---|---|---|---|----|---|---|---|---|---|--|
| 1 | 7 | 2 | 9 | \n | H | a | r | r | y |  |
|---|---|---|---|----|---|---|---|---|---|--|

Dopo la prima invocazione di `nextInt`, rimane in ingresso

|    |   |   |   |   |   |  |
|----|---|---|---|---|---|--|
| \n | H | a | r | r | y |  |
|----|---|---|---|---|---|--|

Invocando `nextLine`, si ottiene una stringa vuota! La soluzione consiste nell’invocare `nextLine` dopo aver letto il numero identificativo dello studente:

```
int studentID = in.nextInt();
in.nextLine(); // consuma il carattere di "fine riga"
String name = in.nextLine();
```

### 11.2.4 Leggere caratteri

Per leggere un carattere per volta, si usa la stringa vuota come *pattern* delimitatore.

```
Scanner in = new Scanner(...);
in.useDelimiter("");
```

Da questo momento in poi ogni invocazione di `next` restituisce una stringa contenente un solo carattere, per cui possiamo elaborare i singoli caratteri in ingresso in questo modo:

```
while (in.hasNext())
{
    char ch = in.next().charAt(0);
    Elabora ch
}
```



### Auto-valutazione

3. Se il flusso d'ingresso contiene i caratteri **6,995.0**, che valore assumono `number` e `input` dopo l'esecuzione di questi enunciati?

```
int number = in.nextInt();
String input = in.next();
```

4. Se il flusso d'ingresso contiene i caratteri **6,995.00 12**, che valore assumono `price` e `quantity` dopo l'esecuzione di questi enunciati?

```
double price = in.nextDouble();
int quantity = in.nextInt();
```

5. Se un file di dati in ingresso contiene una sequenza di numeri, alcuni dei quali sono però assenti e vengono sostituiti dalla stringa **N/A** (*Not Available*, non disponibile), come potete leggere i numeri e ignorare i segnaposto dei valori mancanti?



### Consigli per la produttività 11.1

#### Espressioni canoniche (*regular expressions*)

Le espressioni canoniche descrivono schemi di caratteri (*pattern*). 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 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”, visualizza 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 di righe che corrispondono all'espressione canonica.

Supponiamo di voler cercare tutti i “numeri magici” (Consigli per la qualità 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 utilissimo, perché elenca anche le righe che contengono nomi di variabili, come `x1`. È chiaro che, invece, cerchiamo le sequenze di cifre che *non* seguono immediatamente lettere:

```
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 appena visto, se usate `[^A-Za-z]` come schema, un delimitatore risulta definito come una sequenza di uno o più caratteri che non siano lettere.

Per avere maggiori informazioni in merito alle espressioni canoniche, consultate uno dei tanti siti di Internet che ne parlano, usando come chiave di ricerca “regular expression tutorial”.

## Consigli pratici 11.1

### Elaborare file di testo

L'elaborazione di file di testo che contengano dati reali può essere, sorprendentemente, una vera sfida, per la quale questi consigli vi potranno essere d'aiuto.

Consideriamo questo esempio: leggere due file contenenti dati relativi alle nazioni del mondo, `worldpop.txt` e `worldarea.txt` (li trovate nella cartella `ch11/population` del pacchetto di file scaricabili per questo libro; i file contengono dati relativi alle stesse nazioni, nel medesimo ordine: il primo contiene la popolazione, mentre il secondo contiene la superficie), e scrivere il file `world_pop_density.txt` con lo stesso formato, contenente i nomi delle nazioni, incolonnati a sinistra, e le rispettive densità di popolazione (persone per chilometro quadrato), incolonnate a destra, come in questo esempio:

|                |        |
|----------------|--------|
| Afghanistan    | 50.56  |
| Akrotiri       | 127.64 |
| Albania        | 125.91 |
| Algeria        | 14.18  |
| American Samoa | 288.92 |

#### Fase 1 Analizzate l'elaborazione richiesta

Come sempre, prima di delineare una soluzione, occorre avere ben compreso il compito assegnato. Siete in grado di risolvere il problema a mano (eventualmente con una quantità di dati inferiore)? Se non ci riuscite, approfondite l'analisi.

Un aspetto importante da tenere in considerazione è relativo alla possibilità di elaborare i dati man mano che si rendono disponibili: è, invece, necessario memorizzarli prima tutti? Se, ad esempio, vi viene chiesto di scrivere i dati ordinati, dovete prima averli letti tutti, memorizzandoli in un vettore. Spesso, però, è possibile elaborare i dati “al volo”, senza memorizzarli.

Nel nostro esempio, possiamo leggere i due file una riga per volta e calcolare la densità di popolazione corrispondente a ciascuna riga, perché i dati di popolazione e di superficie sono memorizzati nello stesso ordine.

Ecco lo pseudocodice che descrive l’elaborazione da compiere.

Finché ci sono righe da leggere

Leggi una riga da ciascun file.

Estrai il nome della nazione.

popolazione = numero che segue il nome della nazione nella riga letta dal primo file

area = numero che segue il nome della nazione nella riga letta dal secondo file

Se area != 0

densità = popolazione / area

Scrivi il nome della nazione e la sua densità.

#### Fase 2 Individuate i file da leggere e da scrivere

Queste informazioni dovrebbero risultare evidenti dalla lettura della descrizione del problema. Nel nostro esempio, ci sono due file da leggere, contenenti i dati di popolazione e di superficie, e un solo file da scrivere.

#### Fase 3 Scegliete come acquisire i nomi dei file

Avete quattro possibilità:

- Scrivere i nomi nel codice (ad esempio, "worldpop.txt")
- Chiederli all’utente tramite la console:

```
Scanner in = new Scanner(System.in);
System.out.print("Enter filename: ");
String inFile = in.nextLine();
```

- Usare gli argomenti sulla riga dei comandi (Argomenti avanzati 11.3)
- Usare una finestra di dialogo (Argomenti avanzati 11.1)

Nel nostro esempio, per semplicità scriviamo i nomi nel codice.

#### Fase 4 Decidete se acquisire i dati sotto forma di righe, di parole o di singoli caratteri

Se i dati sono organizzati per righe, di norma è meglio leggere righe. Questo è vero, ad esempio, per dati in forma tabulare, come nel nostro esempio, oppure quando avete bisogno di tener traccia del numero di riga.

Se si devono raccogliere dati che sono distribuiti su più righe, è meglio leggere parole. Ricordatevi, però, che quando leggete parole perdete tutti gli spazi bianchi presenti.

La lettura di singoli caratteri è utile per risolvere quei problemi che richiedono intrinsecamente l'accesso a singoli caratteri: l'analisi della frequenza di caratteri, la sostituzione di caratteri di tabulazione con spazi, la cifratura.

**Fase 5** Nel caso di dati organizzati per righe, estraete le singole componenti

Con il metodo `nextLine` è molto semplice leggere una riga in ingresso, ma dopo occorre estrarre da essa i singoli dati, sotto forma di sottostringhe (Paragrafo 11.2.2).

Per trovare i confini delle sottostringhe, solitamente userete metodi come `Character.isWhitespace` e `Character.isDigit`.

Se alcune delle sottostringhe vi servono sotto forma di numero, le dovete convertire, usando `Integer.parseInt` oppure `Double.parseDouble`.

**Fase 6** Organizzate le caratteristiche e funzionalità comuni in metodi e classi

Spesso l'elaborazione di file prevede compiti ripetitivi: ad esempio, ignorare spazi bianchi o estrarre numeri da stringhe. È veramente utile isolare queste operazioni noiose dal resto del codice.

Nel nostro esempio, c'è un'operazione che viene eseguita due volte: scomporre una riga letta in ingresso nel nome della nazione e nel numero che lo segue. Usando la tecnica vista nel Paragrafo 11.2.2, progettiamo una semplice classe, `CountryValue`, che assolia a questo compito. Ecco il codice sorgente completo.

### File ch11/population/CountryValue.java

```
/*
 * Descrive un valore associato a una nazione.
 */
public class CountryValue
{
    private String country;
    private double value;

    /**
     * Costruisce un esemplare di CountryValue usando una riga.
     * @param line una riga contenente un nome di nazione, seguito
     *             da un valore numerico
     */
    public CountryValue(String line)
    {
        int i = 0; // cerca l'inizio della prima cifra
        while (!Character.isDigit(line.charAt(i))) { i++; }
        int j = i - 1; // cerca la fine della parola precedente
        while (Character.isWhitespace(line.charAt(j))) { j--; }
        country = line.substring(0, j + 1); // estrae il nome della nazione
        value = Double.parseDouble(line.substring(i).trim()); // estrae
  // il valore
    }

    /**
     * Restituisce il nome della nazione.
     * @return il nome della nazione
    }
```

```

        */
    public String getCountry() { return country; }

    /**
     * Restituisce il valore associato.
     * @return il valore associato alla nazione
    */
    public double getValue() { return value; }
}

```

### File ch11/population/PopulationDensity.java

```

import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

public class PopulationDensity
{
    public static void main(String[] args) throws FileNotFoundException
    {
        // apre i file in lettura
        Scanner in1 = new Scanner(new File("worldpop.txt"));
        Scanner in2 = new Scanner(new File("worldarea.txt"));

        // apre il file in scrittura
        PrintWriter out = new PrintWriter("world_pop_density.txt");

        // legge righe da ciascun file
        while (in1.hasNextLine() && in2.hasNextLine())
        {
            CountryValue population = new CountryValue(in1.nextLine());
            CountryValue area = new CountryValue(in2.nextLine());

            // calcola e scrive la densità di popolazione
            double density = 0;
            if (area.getValue() != 0) // evita eventuali divisioni per zero
            {
                density = population.getValue() / area.getValue();
            }
            out.printf("%-40s%15.2f\n", population.getCountry(), density);
        }

        in1.close();
        in2.close();
        out.close();
    }
}

```

## Esempi completi 11.1



### Analizzare nomi di bambini

Nel sito web di un ente pubblico statunitense (*Social Security Administration*) si trovano gli elenchi dei più diffusi nomi per bambini, <http://www.ssa.gov/OACT/babynames/>. In

figura potete vedere il risultato ottenuto interrogando il sistema per conoscere i mille nomi più popolari nel decennio 1990-1999.

I più diffusi nomi per bambini

**Popular Baby Names By Decade**

**Most Popular 1000 Names of the 1990s**

All names are from Social Security card applications for births that occurred in the United States. The data below were extracted from our records at the end of February 2000. See [limitations](#) of such data. The most popular 1000 names of the 1990s were taken from a universe that includes 20,531,547 male births and 19,627,269 female births.

| Male |             |         | Female   |           |         |          |
|------|-------------|---------|----------|-----------|---------|----------|
| Rank | Name        | Number  | Percent* | Name      | Number  | Percent* |
| 1    | Michael     | 462,085 | 2.2506   | Jessica   | 302,962 | 1.5436   |
| 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 | 1.6022   | Sarah     | 224,000 | 1.1413   |
| 5    | Jacob       | 298,016 | 1.4515   | Samantha  | 223,913 | 1.1403   |
| 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   |

Done

Per memorizzare questi dati in forma di testo, basta semplicemente selezionarli e fare "copia e incolla" in un file. Nella cartella ch11/babynames del pacchetto di file scaricabili per questo libro trovate il file **babynames.txt**, che contiene i dati per il decennio 1990-1999.

Ogni riga del file presenta sette dati:

- La posizione in classifica (*rank*, da 1 a 1000)
- Nome maschile in quella posizione, seguito da numero di bambini e percentuale
- Nome femminile in quella posizione, seguito da numero di bambini e percentuale

Ad esempio, la riga

10 Joseph 260365 1.2681 Megan 160312 0.8168

Mostra che il decimo nome più diffuso per bambini maschi fu Joseph, con 260365 nascite, pari a 1.2681% di tutte le nascite di quel decennio. Perché ci furono più Joseph

che Megan? Sembra che i genitori scelgano i nomi femminili in un insieme più vasto, rendendo ciascuno di questi meno frequenti.

Il vostro compito è quello di verificare questa congettura, individuando quali (e, quindi, quanti) nomi figurano nel primo 50 per cento dell'elenco, per i maschi e per le femmine. Semplicemente, visualizzate i nomi di maschi e femmine, insieme alla loro posizione, finché non raggiungete il limite del 50%, separatamente per ciascun genere.

**Fase 1** Analizzate l'elaborazione richiesta

Per elaborare ciascuna linea, innanzitutto leggiamo la posizione in classifica, poi leggiamo i tre valori relativi al nome maschile (nome, conteggio e percentuale), infine ripetiamo quest'ultima attività per il nome femminile. Per interrompere l'elaborazione al raggiungimento del 50 per cento, possiamo sommare le percentuali e fermarci quando, appunto, la somma arriva al 50 per cento.

Ci servono totali separati per maschi e femmine. Quando uno dei totali raggiunge il 50 per cento, smettiamo di scrivere i nomi di quel genere; quando entrambi raggiungono il 50 per cento, smettiamo di leggere.

Ecco lo pseudocodice che descrive l'elaborazione da compiere.

totale Maschile = 0

totale Femminile = 0

Finché (totale Maschile < 50 oppure totale Femminile < 50)

Leggi la posizione in classifica e scrivila.

Leggi il nome maschile, il conteggio e la percentuale.

Se (totale Maschile < 50) scrivi il nome maschile.

Aggiungi la percentuale al totale Maschile.

Ripeti per il genere femminile

**Fase 2** Individuate i file da leggere e da scrivere

Dobbiamo leggere un solo file, `babynames.txt`. Non ci è stato chiesto di scrivere i risultati in un file, quindi li invieremo semplicemente a `System.out`.

**Fase 3** Scegliete come acquisire i nomi dei file

Non abbiamo bisogno di chiedere nomi di file all'utente.

**Fase 4** Decidete se acquisire i dati sotto forma di righe, di parole o di singoli caratteri

I dati forniti dall'ente statunitense non prevedono nomi contenenti spazi, come "Mary Jane", quindi ogni riga del file contiene esattamente sette dati: due stringhe prive di spazi e cinque numeri. Dati di questo tipo si possono tranquillamente leggere sotto forma di parole e numeri.

**Fase 5** Nel caso di dati organizzati per righe, estraete le singole componenti

Possiamo saltare questa fase, perché non leggiamo riga per riga.

Supponiamo, però, di aver deciso, nella Fase 4, di leggere i dati riga per riga. In tal caso, dovremmo scomporre ciascun riga in sette stringhe, convertendone cinque in numeri: sarebbe piuttosto noioso e sarebbe meglio tornare sui nostri passi.

#### Fase 6 Organizzate le caratteristiche e funzionalità comuni in metodi e classi

Nello pseudocodice abbiamo scritto: Ripeti per il genere femminile. È quindi evidente che esistono attività comuni, ripetute, che spingono all'utilizzo di un metodo ausiliario, per le tre fasi seguenti:

- Leggi il nome, il conteggio e la percentuale.
- Visualizza il nome se il totale è inferiore al 50 per cento.
- Aggiungi la percentuale al totale.

Per questo scopo usiamo la classe ausiliare `RecordReader`, di cui costruiamo due esemplari, uno per i nomi maschili e uno per i nomi femminili: ciascuno dei due gestisce un totale distinto, aggiornandolo con la percentuale letta di volta in volta, e visualizza nomi fino al raggiungimento della soglia prevista. Il ciclo principale dell'elaborazione diventa quindi

```
RecordReader boys = new RecordReader(LIMIT);
RecordReader girls = new RecordReader(LIMIT);

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 un dato completo (nome, conteggio e percentuale)
e visualizza il nome se il totale è inferiore alla soglia.
@param in il flusso da cui leggere
*/
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;
}
```

mentre nel seguito trovate il codice sorgente completo.

Osservate come il programma renda evidente un risultato notevole dell'analisi: nel decennio esaminato, per assegnare un nome alla metà di tutti i nati sono stati usati solamente 69 nomi maschili e 153 nomi femminili. Una buona notizia per chi produce articoli per la festa del papà, personalizzandoli con il nome! L'Esercizio P11.8 vi chiederà di studiare come questa distribuzione sia cambiata negli anni.

### File ch11/babynames/BabyNames.java

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class BabyNames
{
    public static final double LIMIT = 50;

    public static void main(String[] args) throws FileNotFoundException
    {
        Scanner in = new Scanner(new File("babynames.txt"));

        RecordReader boys = new RecordReader(LIMIT);
        RecordReader girls = new RecordReader(LIMIT);

        while (boys.hasMore() || girls.hasMore())
        {
            int rank = in.nextInt();
            System.out.print(rank + " ");
            boys.process(in);
            girls.process(in);
            System.out.println();
        }

        in.close();
    }
}
```

### File ch11/babynames/RecordReader.java

```
import java.util.Scanner;

/**
 * Questa classe elabora dati relativi a un nome.
 */
public class RecordReader
{
    private double total;
    private double limit;

    /**
     * Costruisce un esemplare di RecordReader
     * con totale zero.
     */
    public RecordReader(double aLimit)
    {
        total = 0;
```

```

        limit = aLimit;
    }

    /**
     * Legge un dato completo (nome, conteggio e percentuale)
     * e visualizza il nome se il totale è inferiore alla soglia.
     * @param in il flusso da cui leggere
    */
    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;
    }

    /**
     * Verifica se occorre leggere altri dati.
     * @return true se la soglia non è stata ancora raggiunta
    */
    public boolean hasMore()
    {
        return total < limit;
    }
}

```

## 11.3 Lanciare eccezioni

Esistono due aspetti relativi alla gestione delle eccezioni: *segnalazione* e *ripristino*. Una delle cose più difficili nella gestione degli errori è che, solitamente, il punto in cui viene segnalato l'errore non è vicino al punto in cui si ripristina la situazione corretta. Ad esempio, il metodo `get` della classe `ArrayList` può individuare che si sta effettuando un accesso a un elemento inesistente, ma non ha sufficienti informazioni per decidere cosa fare in tale situazione d'errore. Si dovrebbe chiedere all'utente di tentare una diversa operazione, oppure è meglio terminare bruscamente il programma dopo aver salvato il lavoro dell'utente? Queste decisioni vanno prese in una zona diversa del programma.

In Java, la *gestione delle eccezioni* è un meccanismo flessibile per trasferire il controllo dell'esecuzione del programma dal punto in cui viene segnalato l'errore a un gestore competente per il ripristino di una situazione corretta. La parte restante di questo capitolo discute in dettaglio il meccanismo per la gestione delle eccezioni.

Quando individuate una condizione d'errore, il vostro compito è semplice: *lanciate* (*throw*) un oggetto appropriato di tipo eccezione e non dovete fare altro. Ad esempio, supponete che qualcuno cerchi di prelevare troppi soldi da un conto bancario.

Per segnalare una condizione eccezionale si usa l'enunciato `throw`, lanciando un oggetto eccezione.

```

public class BankAccount
{
    ...
    public void withdraw(double amount)
    {
        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 tutti i tipi di condizioni eccezionali: la Figura 1 mostra quelle più utili.

Cercate un'eccezione che potrebbe descrivere la vostra situazione. Che ne dite di `IllegalStateException`? Il conto bancario si trova in uno stato che non consente l'operazione di prelievo? In realtà non è così, alcune operazioni di prelievo potrebbero avere successo. È l'argomento a non essere valido? Certo, ha un valore troppo elevato, quindi lanciamo un oggetto di tipo `IllegalArgumentException` (la parola *argomento* è, in questo contesto, sinonimo di *valore di un parametro*)

```
public class BankAccount
{
    public void withdraw(double amount)
    {
        if (amount > balance)
        {
            throw new IllegalArgumentException("Amount exceeds balance");
        }
        balance = balance - amount;
    }
    ...
}
```

L'enunciato

```
throw new IllegalArgumentException("Amount exceeds balance");
```

costruisce un oggetto di tipo `IllegalArgumentException` e lo lancia.

## Sintassi di Java

### 11.1 Lanciare un'eccezione

#### Sintassi

```
throw oggettoEccezione;
```

#### Esempio

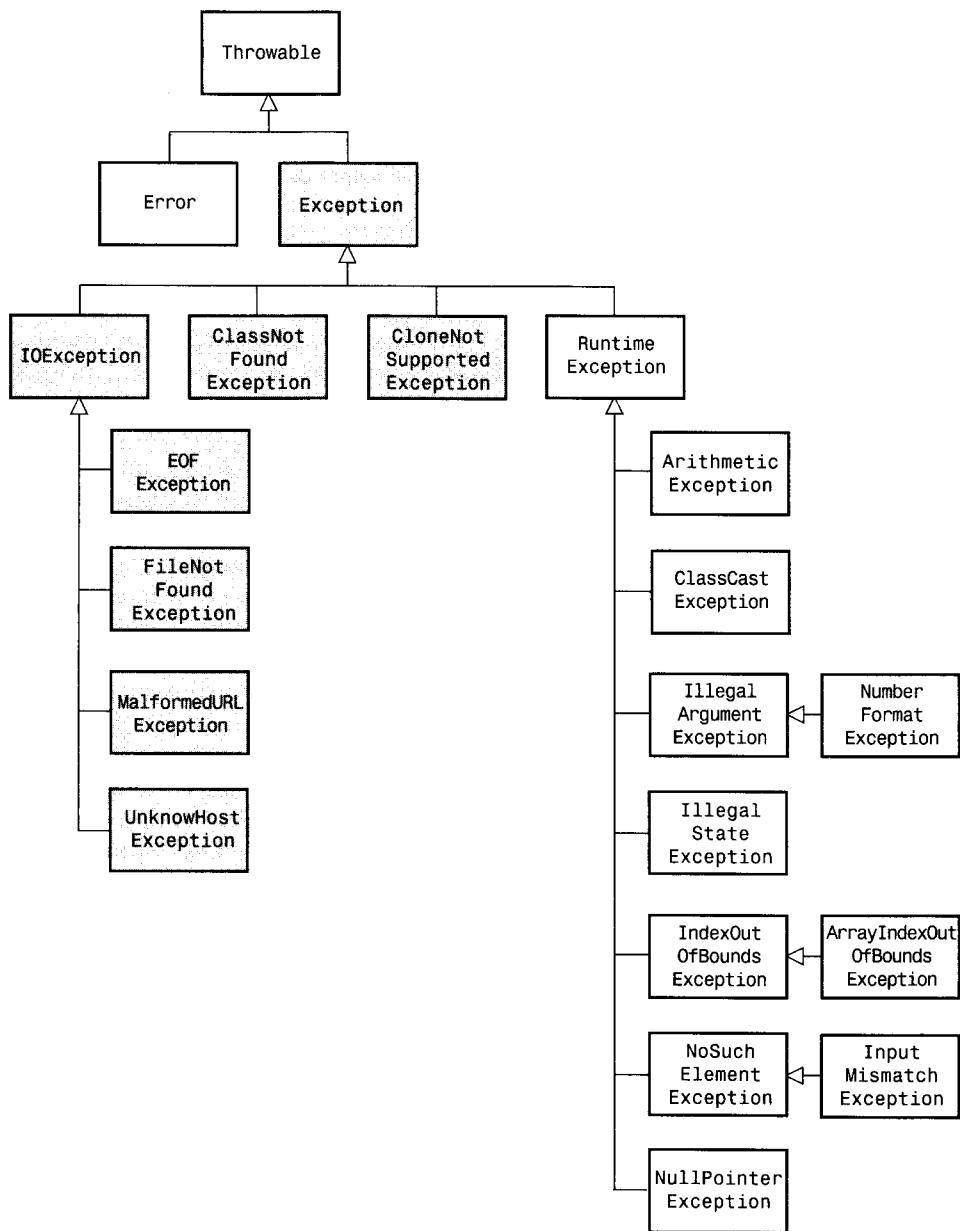
Si costruisce un nuovo oggetto di tipo eccezione, poi lo si lancia.

```
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.

**Figura 1**  
La gerarchia delle classi di eccezioni.



Quando lanciate un'eccezione, il metodo termina immediatamente la propria esecuzione.

Quando lanciate un'eccezione, l'esecuzione non procede con l'enunciato successivo ma passa a un gestore dell'eccezione, di cui per ora non ci occupiamo: sarà argomento del Paragrafo 11.5.



## Auto-valutazione

6. Come si può modificare il metodo `deposit` per essere certi che il saldo non diventi mai negativo?
7. Immaginate di costruire un nuovo conto bancario con saldo zero e di invocare `withdraw(10)`: qual è, poi, il valore di `balance`?

## 11.4 Eccezioni controllate e non controllate

Esistono due tipi di eccezioni:  
**controllate** e **non controllate**.  
Le eccezioni non controllate  
estendono la classe  
**RuntimeException**  
o **Error**.

In Java le eccezioni ricadono entro due categorie, eccezioni *a controllo obbligatorio* (“checked”, in breve: *controllate*) e *a controllo non obbligatorio* (“unchecked”, in breve: *non controllate*). Quando invocate un metodo che lancia un’eccezione controllata, il compilatore verifica che non venga ignorata: *dovete* dichiarare cosa avete intenzione di fare se viene lanciata. Ad esempio, tutte le sottoclassi di `IOException` sono eccezioni controllate. Al contrario, il compilatore non richiede che vi occupiate necessariamente delle eccezioni non controllate, come `NumberFormatException`, `IllegalArgumentException` o `NullPointerException`. Più in generale, tutte le eccezioni che appartengono a sottoclassi di `RuntimeException` sono a controllo non obbligatorio, mentre tutte le altre sottoclassi della classe `Exception` lo sono (nella Figura 1 le eccezioni a controllo obbligatorio sono evidenziate in grigio). Esiste una seconda categoria di errori interni, segnalati lanciando oggetti di tipo `Error`. Un esempio è `OutOfMemoryError`, che viene lanciato quando è stata usata tutta la memoria disponibile. Questi sono errori fatali che accadono di rado e non ricadono sotto il vostro controllo: anch’essi sono a controllo non obbligatorio.

Perché ci sono due tipi di eccezioni? Un’eccezione a controllo obbligatorio descrive un problema che prima o poi può accadere, indipendentemente da quanto siete attenti. Le eccezioni a controllo non obbligatorio, al contrario, accadono per un vostro errore. Ad esempio, la fine inattesa di un file può essere provocata da forze che non sono sotto il vostro controllo, come un errore del disco o l’interruzione di un collegamento di rete, ma siete da biasimare per un `NullPointerException`, perché è il vostro codice a essere in errore, cercando di usare un riferimento `null`.

Il compilatore non verifica che le eccezioni di tipo `NullPointerException` vengano gestite, perché, invece di prevedere l’uso di un gestore per tale eccezione, dovreste verificare che i vostri riferimenti non siano `null` prima di usarli. Il compilatore insiste, invece, perché il vostro programma sia in grado di gestire quelle condizioni di errore che non potete prevenire.

In realtà, queste categorie non sono perfette. Ad esempio, se un utente inserisce un dato che non sia un numero intero, il metodo `Scanner.nextInt` lancia `InputMismatchException`, che è un’eccezione a controllo non obbligatorio: dal momento che il programmatore non può impedire che l’utente inserisca dati sbagliati, sarebbe stato più appropriato usare un’eccezione a controllo obbligatorio (i progettisti della classe `Scanner` hanno fatto questa scelta per rendere più agevole l’utilizzo della classe da parte dei programmatori inesperti).

Come potete notare osservando la Figura 1, la maggior parte delle eccezioni controllate vengono utilizzate nella gestione dei dati in ingresso o in uscita, che è un fertile terreno per guasti esterni che non sono sotto il vostro controllo: un file può essere stato

Le eccezioni controllate sono dovute  
a circostanze esterne che  
il programmatore non può evitare  
e il compilatore verifica che  
il programma le gestisca.

rimosso o corrotto, una connessione di rete può essere sovraccarica, un server può non essere disponibile, e così via. Quindi, avrete bisogno di gestire eccezioni controllate principalmente quando programmate usando file e flussi.

Avete visto come usare la classe `Scanner` anche per leggere dati da un file, passando al suo costruttore un oggetto di tipo `File`:

```
String filename = ...;
File inFile = new File(filename);
Scanner in = new Scanner(inFile);
```

Questo costruttore di `Scanner`, però, può lanciare `FileNotFoundException`, che è un'eccezione controllata, per cui dovete necessariamente dire al compilatore cosa avete intenzione di fare. Avete due possibilità: potete usare le tecniche che vedrete nel Paragrafo 11.5, oppure potete dire semplicemente al compilatore che siete consapevoli di questa eccezione e che volete che il vostro metodo termini la sua esecuzione quando essa viene lanciata. Il metodo che legge i dati in ingresso raramente sa come comportarsi in caso di errori inattesi, per cui la seconda è solitamente la scelta migliore.

Per dichiarare che un metodo dovrebbe terminare nel caso in cui venga lanciata al suo interno un'eccezione a controllo obbligatorio, si contrassegna il metodo stesso con la clausola `throws`:

```
public void read(String filename) throws FileNotFoundException
{
    File inFile = new File(filename);
    Scanner in = new Scanner(inFile);
    ...
}
```

**Aggiungete la clausola `throws` a un metodo che può lanciare un'eccezione a controllo obbligatorio.**

La clausola `throws` segnala a chi invoca il vostro 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 dire a chi l'ha invocato che essa può essere lanciata dal metodo.

Se il vostro metodo può lanciare più eccezioni a controllo obbligatorio, separate con virgoletti i nomi delle relative classi:

```
public void read(String filename)
    throws FileNotFoundException, NoSuchElementException
```

Ricordate sempre che le classi che rappresentano eccezioni costituiscono una gerarchia di ereditarietà. Ad esempio, `FileNotFoundException` è una sottoclasse di `IOException`. Di conseguenza, se un metodo può lanciare sia `FileNotFoundException` sia `IOException`, potete contrassegnarlo solamente con `IOException`.

Non gestire un'eccezione quando sapete che essa può accadere sembra un comportamento in qualche modo irresponsabile, ma, in realtà, se non sapete come rimediare al problema, non catturare un'eccezione è solitamente la cosa migliore. Dopo tutto, cosa potete fare in un metodo `read` che agisce a un basso livello di astrazione? Potete colloquiare con l'utente? Come? Inviandogli un messaggio tramite `System.out`? Non sapete se questo metodo sia stato invocato all'interno di un programma grafico o, addirittura, in un sistema "embedded", come un distributore automatico, dove l'utente

## Sintassi di Java

### 11.2 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.

non potrà mai vedere `System.out`. E, anche se i vostri utenti possono vedere il vostro messaggio d'errore, come potete sapere se sono in grado di capire la lingua inglese? La vostra classe può essere usata per costruire un'applicazione per utenti di un altro paese. Se non potete contattare l'utente, siete in grado di riparare i dati e proseguire? Come? Se impostate una variabile al valore zero o `null`, oppure usate una stringa vuota, ciò può semplicemente ritardare il fallimento del programma, che accadrà in modo ancor più misterioso.

Ovviamente, alcuni metodi del programma sanno come comunicare con l'utente o come intraprendere altre azioni di recupero: consentendo all'eccezione di raggiungere tali metodi, consentite la sua gestione da parte di un gestore competente.



#### Auto-valutazione

8. Immaginate che un metodo invochi il costruttore di `Scanner`, che può lanciare un'eccezione di tipo `FileNotFoundException`, e il metodo `nextInt` della classe `Scanner`, che può lanciare un'eccezione di tipo `NoSuchElementException` oppure `InputMismatchException`. Quali eccezioni dovrebbero essere elencate nella clausola `throws`?
9. Perché `NullPointerException` è un'eccezione a controllo non obbligatorio?

### 11.5 Catturare eccezioni

In un metodo che è in grado di gestire un particolare tipo di eccezione, inserite gli enunciati che possono lanciare l'eccezione all'interno di un blocco `try`, seguito dal gestore posto all'interno di 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 vostro programma termina. Però, non si vuole che un programma scritto professionalmente termini bruscamente soltanto perché qualche metodo ha individuato una condizione d'errore imprevista, quindi dovete installare gestori di eccezioni per tutte le eccezioni che possono essere lanciate dal vostro programma.

Un gestore di eccezione si installa con l'enunciato **try/catch**. Ciascun blocco **try** contiene uno o più enunciati che possono provocare il lancio di un'eccezione ed è seguito da clausole **catch**, ognuna contenente il gestore di uno specifico tipo di eccezione.

Ecco un esempio:

```
try
{
    String filename = ...;
    File inFile = new File(filename);
    Scanner in = new Scanner(inFile);
    String input = in.next();
    int value = Integer.parseInt(input);
    ...
}
catch (IOException exception)
{
    exception.printStackTrace();
}
catch (NumberFormatException exception)
{
    System.out.println("Input was not a number");
}
```

In questo blocco **try** possono essere lanciate eccezioni di tre tipi diversi: il costruttore di **Scanner** può lanciare **FileNotFoundException**, il metodo **Scanner.next** può lanciare **NoSuchElementException** e il metodo **Integer.parseInt** può lanciare **NumberFormatException**.

Se qualcuna 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** (ricordate che **FileNotFoundException** è una sottoclasse di **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 attiva finché non viene catturata da un altro blocco **try** oppure finché arriva a provocare la terminazione brusca del metodo **main**, cioè del programma.

Se viene eseguito il blocco della clausola **catch (IOException exception)**, significa che qualcuno dei metodi invocati all'interno del blocco **try** ha lanciato un oggetto eccezione di tipo **IOException**, il cui riferimento viene memorizzato nella variabile **exception**. La clausola **catch** può, quindi, esaminare tale oggetto per identificare maggiori dettagli sul guasto. Ad esempio, potete visualizzare un elenco della catena di invocazioni di metodi che ha portato all'eccezione, invocando

```
exception.printStackTrace()
```

**Sintassi di Java****11.3 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`.

```
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");
}
```

Questo costruttore può lanciare `FileNotFoundException`.

Questa è l'eccezione che è stata lanciata.

`FileNotFoundException` è un caso speciale di `IOException`.

Nelle clausole `catch` usate come esempio ci siamo limitati a informare l'utente della causa del problema, mentre un modo migliore per gestire tali eccezioni consiste nel dare all'utente un'altra possibilità di inserire correttamente i dati, come vedrete nel Paragrafo 11.8. È importante ricordare che dovreste inserire clausole `catch` soltanto dove potete gestire con competenza un particolare tipo di eccezione.

**Auto-valutazione**

10. Seguite passo dopo passo il flusso di esecuzione del blocco `try` visto in questo paragrafo nel caso in cui il file di cui viene assegnato il nome esista ma sia vuoto.
11. C'è differenza tra catturare un'eccezione a controllo obbligatorio e catturare un'eccezione a controllo non obbligatorio?

**Consigli per la qualità 11.1****Lanciare presto, catturare tardi**

Quando un metodo si trova di fronte a 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 non contenga, invece, un numero: usare semplicemente un 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.

Lanciare un'eccezione non appena si riscontra un problema, ma catturarla soltanto quando il problema può essere risolto o gestito..

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.

Questi principi possono essere riassunti in un motto: "lanciare presto, catturare tardi".

## Consigli per la qualità 11.2

### Non mettete a tacere le eccezioni

Quando invocate un metodo che lancia un'eccezione a controllo obbligatorio e non avete definito il gestore corrispondente, il compilatore protesta. Nell'ansia di portare a termine il vostro lavoro, è comprensibile che zittiate il compilatore *mettendo a tacere* ("squelching") l'eccezione, ad esempio in questo modo:

```
try
{
    File inFile = new File(filename);
    Scanner in = new Scanner(inFile);
    // il compilatore protestava per FileNotFoundException
}
catch (Exception 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 segnalare un problema a un gestore competente: l'inserimento di un gestore incompetente nasconde semplicemente una condizione d'errore che potrebbe essere seria.

## 11.6 La clausola `finally`

A volte avrete bisogno di intraprendere alcune azioni indipendentemente dal fatto che un'eccezione sia stata lanciata oppure no: il costrutto `finally` viene usato proprio per gestire una situazione simile. Ecco una situazione tipica.

È importante chiudere gli oggetti di tipo `PrintWriter`, per essere certi che tutti i dati vengano scritti effettivamente nel file. Nel frammento di codice seguente, apriamo un flusso per scrivere dati in un file, invochiamo uno o più metodi, quindi chiudiamo il file:

```
PrintWriter out = new PrintWriter(filename);
writeData(out);
out.close(); // può darsi che non si arrivi mai qui
```

Ora, supponete che uno dei metodi che precedono l'ultima linea lanci un'eccezione: in questo caso l'invocazione del metodo `close` non viene mai eseguita! Risolvete questo problema inserendo l'invocazione di `close` all'interno di una clausola `finally`:

```
PrintWriter out = new PrintWriter(filename);
try
{
    writeData(out);
}
finally
{
    out.close();
}
```

Una volta che è iniziata l'esecuzione di un blocco `try`, si ha la garanzia che gli enunciati di una clausola `finally` verranno eseguiti, indipendentemente dal fatto che venga lanciata un'eccezione oppure no.

Nel caso di esecuzione normale, non ci saranno problemi: quando il blocco `try` sarà terminato, verrà eseguita la clausola `finally`, che provvede a chiudere il file. Se, invece, viene lanciata un'eccezione, la clausola `finally` viene comunque eseguita, prima di trasferire l'eccezione al suo gestore.

Usate la clausola `finally` ogni volta che avete bisogno di fare un po' di pulizia, ad esempio chiudendo un file, per essere sicuri che la pulizia venga eseguita indipendentemente da come termina il metodo.

## Sintassi di Java

### 11.4 La clausola `finally`

#### Sintassi

```
try
{
    enunciato
    enunciato
    ...
}
finally
{
    enunciato
    enunciato
    ...
}
```

#### Esempio

Questa variabile deve essere dichiarata al di fuori del blocco `try`, in modo che sia accessibile dall'interno della clausola `finally`.

Questo codice può lanciare eccezioni.

Questo codice viene sempre eseguito, anche se viene lanciata un'eccezione.

```
PrintWriter out = new PrintWriter(filename);
try
{
    writeData(out);
}
finally
{
    out.close();
}
```

Si può anche inserire una clausola `finally` dopo una o più clausole `catch`. In questo caso, il codice della clausola `finally` viene eseguito al termine del blocco `try`, in una delle seguenti situazioni:

1. Dopo aver portato a termine l'ultimo enunciato del blocco `try`.
2. Dopo aver portato a termine l'ultimo enunciato di una clausola `catch` che abbia catturato un'eccezione lanciata nel blocco `try`.
3. Quando nel blocco `try` è stata lanciata un'eccezione che non viene catturata.

In ogni caso, vi raccomandiamo di non usare clausole `catch` e `finally` nel medesimo blocco `try`, come vedrete nei Consigli per la qualità 11.3.



## Auto-valutazione

12. Perché la variabile `out` è stata dichiarata al di fuori del blocco `try`?
13. Seguite passo dopo passo il flusso di esecuzione del blocco `try` visto in questo paragrafo nel caso in cui il file di cui viene assegnato il nome non esista.



## Consigli per la qualità 11.3

### Non usate `catch` e `finally` nel medesimo blocco `try`

Si può avere la tentazione di combinare clausole `catch` e `finally`, ma il codice che ne risulta è di difficile comprensione, per cui dovreste, invece, usare un enunciato `try/finally` per chiudere le risorse utilizzate e un enunciato `try/catch` separato per gestire gli errori, come in questo esempio:

```
try
{
    PrintWriter out = new PrintWriter(filename);
    try
    {
        Scrittura dei dati in out
    }
    finally
    {
        out.close();
    }
}
catch (IOException exception)
{
    Gestione dell'eccezione
}
```

Notate che gli enunciati annidati funzionano correttamente anche nel caso in cui il costruttore di `PrintWriter` lanci un'eccezione, come si può vedere nell'Esercizio R11.18.



## Argomenti avanzati 11.4

### Gestione automatica delle risorse in Java 7

In Java 7, avete a disposizione una nuova forma di blocco `try` che provvede a chiudere automaticamente oggetti che implementino l'interfaccia `Closeable`, come quelli di tipo `PrintWriter` o `Scanner`.

```
try (PrintWriter out = new PrintWriter(filename))
{
    Scrittura dei dati in out
}
```

Quando termina l'esecuzione del blocco `try`, viene automaticamente invocato il metodo `close` con l'oggetto `out`, indipendentemente dal lancio di un'eccezione: non è necessaria la presenza di una clausola `finally`.

## 11.7 Progettare eccezioni

A volte nessuno dei tipi di eccezioni standard descrive abbastanza bene la particolare condizione di errore che vi interessa: in tal caso, potete progettare una vostra classe d'eccezione. Considerate un conto bancario: quando si tenta di prelevare una somma superiore al saldo, segnaliamo 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.

In questi casi, è bene decidere con cura quale classe estendere, tra quelle già esistenti nella gerarchia, scegliendo quella più appropriata: 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: un costruttore senza argomenti e uno che accetta una stringa come messaggio che descrive il motivo dell'eccezione. Ecco la dichiarazione della classe di eccezioni.

```
public class InsufficientFundsException extends RuntimeException
{
    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

14. Qual è lo scopo dell'invocazione `super(message)` nel secondo costruttore di `InsufficientFundsException`?
15. Immaginate di leggere da un file i dati relativi a conti bancari. Se, contrariamente alle vostre attese, vi trovate di fronte a un valore che non è di tipo `double`, decidete di utilizzare una eccezione che chiamate `BadDataException`. Quale classe dovrebbe estendere?



### Consigli per la qualità 11.4

#### 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 di tipo `RuntimeException`, perché ciò renderebbe molto più complessa la cattura dell'eccezione. Se, infatti, catturate tutte le eccezioni di tipo `RuntimeException`, la vostra clausola `catch` verrebbe attivata anche da eccezioni come `NullPointerException`, `ArrayIndexOutOfBoundsException` e così via. Dovreste allora esaminare attentamente l'oggetto che rappresenta l'eccezione, tentando di capire se l'eccezione 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.

## 11.8 Un esempio completo

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 per questo programma potrebbe essere simile a questo:

```
3
1.45
-2.1
0.05
```

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. Dobbiamo soltanto essere sicuri che i metodi che elaborano i valori in ingresso lancino 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 di tipo `FileNotFoundException`, che è perfetta per tale situazione. Infine, quando il file di dati è in un formato non corretto, lanceremo un'eccezione a controllo obbligatorio progettata da noi, `BadDataException`. Usiamo un'eccezione controllata perché la corruzione dei dati in un file sfugge al controllo del programmatore.

Chi può porre rimedio agli errori segnalati da tali eccezioni? Il metodo `main` del programma `DataAnalyzer` è l'unico ad 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 ch11/data/DataAnalyzer.java

```

import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;

/**
 * Questo programma legge un file contenente numeri e ne analizza
 * il contenuto. Se il file non esiste o contiene stringhe che non siano
 * numeri, visualizza un messaggio d'errore.
 */

public class DataAnalyzer
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        DataSetReader reader = new DataSetReader();

        boolean done = false;
        while (!done)
        {
            try
            {
                System.out.println("Please enter the file name: ");
                String filename = in.next();

                double[] data = reader.readFile(filename);
                double sum = 0;
                for (double d : data) sum = sum + d;
                System.out.println("The sum is " + sum);
                done = true;
            }
            catch (FileNotFoundException exception)

```

```
        System.out.println("File not found.");
    }
    catch (BadDataException exception)
    {
        System.out.println("Bad data: " + exception.getMessage());
    }
    catch (IOException exception)
    {
        exception.printStackTrace();
    }
}
```

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 con il file in lettura, l'eccezione viene semplicemente trasferita all'invocante.

```
public double[] readFile(String filename) throws IOException
{
    File inFile = new File(filename);
    Scanner in = new Scanner(inFile);
    try
    {
        readData(in);
        return data;
    }
    finally
    {
        in.close();
    }
}
```

Il metodo dichiara di lanciare `IOException`, la superclasse comune di `FileNotFoundException` (lanciata dal costruttore di `Scanner`) e `BadDataException` (lanciata dal metodo `readData`).

Ecco 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 le 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 un'eccezione di tipo `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 eseguito la clausola `finally` che chiude l'oggetto di tipo `Scanner`.
8. Per le eccezioni di tipo `BadDataException`, `DataAnalyzer.main` ha un gestore che visualizza un messaggio all'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 ch11/data/DataSetReader.java

```

import java.io.File;
import java.io.IOException;
import java.util.Scanner;

```

```
/**  
 * Legge un insieme di dati da un file, che deve avere questo formato:  
 * numeroDiValori  
 * valore1  
 * valore2  
 * ...  
 */  
public class DataSetReader  
{  
    private double[] data;  
  
    /**  
     * Legge un insieme di dati.  
     * @param filename il nome del file che contiene i dati  
     * @return i dati presenti nel file  
     */  
    public double[] readFile(String filename) throws IOException  
    {  
        File inFile = new File(filename);  
        Scanner in = new Scanner(inFile);  
        try  
        {  
            readData(in);  
            return data;  
        }  
        finally  
        {  
            in.close();  
        }  
    }  
  
    /**  
     * Legge tutti i dati.  
     * @param in lo scanner da cui 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");  
    }  
  
    /**  
     * Legge uno dei valori.  
     * @param in lo scanner da cui leggere  
     * @param i la posizione del valore da leggere  
     */  
}
```

```

private void readValue(Scanner in, int i) throws BadDataException
{
    if (!in.hasNextDouble())
        throw new BadDataException("Data value expected");
    data[i] = in.nextDouble();
}
}

```



## Note di cronaca 11.1

### 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-distruzione: 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, usata 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 dei dispositivi genera un'eccezione se un numero in virgola mobile troppo grande viene convertito in un intero, ma, sfortunatamente, i programmatore del dispositivo avevano deciso che tale situazione non sarebbe mai accaduta e non avevano fornito 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 imma-

ginato 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 espliciti ai programmatore, una cosa a cui dovete pensare quando state maledicendo il compilatore Java perché si lamenta di eccezioni non catturate.

### File ch11/data/BadDataException.java

```
import java.io.IOException;

/**
 * Questa classe segnala un errore nei dati in ingresso.
 */
public class BadDataException extends IOException
{
    public BadDataException() {}
    public BadDataException(String message)
    {
        super(message);
    }
}
```



### Auto-valutazione

16. Perché il metodo `DataSetReader.readFile` non cattura alcuna eccezione?
17. Seguite passo dopo passo il flusso di esecuzione nel caso in cui l'utente specifichi un file che esiste ma è vuoto.

## 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`.
- Quando avete finito di scrivere in un flusso, dovete chiuderlo.

### Scegliere la strategia più appropriata per l'elaborazione di dati in ingresso

- Il metodo `next` legge una parola per volta; per specificare uno schema per suddividere le parole si invoca `Scanner.useDelimiter`.
- Il metodo `nextLine` legge una riga in ingresso e consuma il carattere di “nuova riga” che si trova alla fine della riga letta.
- I metodi `nextInt` e `nextDouble` consumano eventuali spazi bianchi e leggono il numero che segue.
- Per leggere un carattere per volta, si usa la stringa vuota come *pattern* delimitatore.

### Capire quando e come lanciare un'eccezione

- Per segnalare una condizione eccezionale si usa l'enunciato `throw`, lanciando un oggetto eccezione.
- Quando lanciate un'eccezione, il metodo termina immediatamente la propria esecuzione.

### Scegliere tra eccezioni controllate e non controllate

- Esistono due tipi di eccezioni: *controllate* e *non controllate*. Le eccezioni non controllate estendono la classe `RuntimeException` o `Error`.
- Le eccezioni controllate 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.

### Usare i gestori di eccezione per disaccoppiare la rilevazione d'errore e la sua segnalazione

- In un metodo che è in grado di gestire un particolare tipo di eccezione, inserite gli enunciati che possono lanciare l'eccezione all'interno di un blocco `try`, seguito dal gestore posto all'interno di una clausola `catch`.
- Lanciare un'eccezione non appena si riscontra un problema, ma catturarla soltanto quando il problema può essere risolto o gestito.

### Usare la clausola `finally` per garantire una corretta gestione delle risorse in caso di lancio di eccezione

- Una volta che è iniziata l'esecuzione di un blocco `try`, si ha la garanzia che gli enunciati di una clausola `finally` verranno eseguiti, indipendentemente dal fatto che venga lanciata un'eccezione oppure no.

### Progettare eccezioni per descrivere condizioni di errore specifiche

- Per descrivere una condizione d'errore, progettate una sottoclasse di una classe di eccezione esistente.

## Classi, oggetti e metodi presentati nel capitolo

|                                                 |                                               |
|-------------------------------------------------|-----------------------------------------------|
| <code>java.io.EOFException</code>               | <code>java.lang.RuntimeException</code>       |
| <code>java.io.File</code>                       | <code>java.lang.Throwable</code>              |
| <code>java.io.FileNotFoundException</code>      | <code>getMessage</code>                       |
| <code>java.io.FileReader</code>                 | <code>printStackTrace</code>                  |
| <code>java.io.IOException</code>                | <code>java.util.NoSuchElementException</code> |
| <code>java.io.PrintWriter</code>                | <code>java.util.Scanner</code>                |
| <code>    close</code>                          | <code>close</code>                            |
| <code>java.lang.Error</code>                    | <code>javax.swing.JFileChooser</code>         |
| <code>java.lang.IllegalArgumentException</code> | <code>getSelectedFile</code>                  |
| <code>java.lang.IllegalStateException</code>    | <code>showOpenDialog</code>                   |
| <code>java.lang.NullPointerException</code>     | <code>showSaveDialog</code>                   |
| <code>java.lang.NumberFormatException</code>    |                                               |

## Esercizi di ripasso

- \*\* **Esercizio R11.1.** Cosa succede se cercate di aprire per la lettura un file inesistente? Che cosa succede se cercate di aprire per la scrittura un file inesistente?
- \*\*\* **Esercizio 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*)? Dimostratelo con un breve programma di prova.
- \* **Esercizio R11.3.** Come si apre un file il cui nome contiene una barra rovesciata, come `c:\temp\output.dat`?
- \*\*\* **Esercizio R11.4.** Cos’è la riga dei comandi? Come fa un programma a leggere gli argomenti dalla riga dei comandi?
- \*\* **Esercizio R11.5.** Fornite due esempi di programmi del vostro computer che leggono argomenti dalla riga dei comandi.
- \*\* **Esercizio R11.6.** Se il programma `Woozle` viene eseguito con il comando

```
java Woozle -Dname=piglet -I\eyore -v heff.txt a.txt lump.txt
```

quali sono i valori di `args[0]`, `args[1]` e così via?

- \*\* **Esercizio R11.7.** Qual è la differenza tra lanciare e catturare un'eccezione?
- \*\* **Esercizio R11.8.** Cos'è un'eccezione a controllo obbligatorio? Cos'è un'eccezione a controllo non obbligatorio? L'eccezione di tipo `NullPointerException` è a controllo obbligatorio oppure no? Quali eccezioni siete obbligati a dichiarare con una clausola `throws`?
- \* **Esercizio R11.9.** Perché non è obbligatorio dichiarare che un metodo può lanciare un'eccezione di tipo `NullPointerException`?
- \*\* **Esercizio R11.10.** Quando un programma esegue un enunciato `throw`, qual è il successivo enunciato che viene eseguito?
- \* **Esercizio R11.11.** Cosa succede se non esiste la clausola `catch` corrispondente a un'eccezione che viene lanciata?
- \* **Esercizio R11.12.** Cosa può fare un programma con l'oggetto eccezione ricevuto da una clausola `catch`?
- \* **Esercizio R11.13.** Il tipo dell'oggetto eccezione è sempre uguale al tipo dichiarato nella clausola `catch` che lo cattura?
- \* **Esercizio R11.14.** Che tipo di oggetti potete lanciare? Potete lanciare una stringa? È un numero intero?
- \*\* **Esercizio R11.15.** A cosa serve la clausola `finally`? Fornite un esempio di come poterla usare.
- \*\*\* **Esercizio R11.16.** Cosa succede quando viene lanciata un'eccezione, viene eseguito il codice di una clausola `finally` e questo, a sua volta, lancia un'eccezione di un tipo diverso dalla prima? Quale eccezione viene catturata da una clausola `catch` circostante? Scrivete un programma d'esempio e provate.
- \*\* **Esercizio R11.17.** Quali eccezioni possono essere lanciate dai metodi `next` e `nextInt` della classe `Scanner`? Sono eccezioni controllate o non controllate?
- \*\*\* **Esercizio R11.18.** Supponete che il codice riportato in Consigli per la qualità 11.3 sia stato condensato in un unico enunciato `try/catch/finally`:

```
PrintWriter out = new PrintWriter(filename);
try
{
    Scrittura dei dati in out
}
catch (IOException exception)
{
    Gestione dell'eccezione
}
finally
{
    out.close();
}
```

Identificare lo svantaggio che caratterizza questa versione (*suggerimento*: cosa succede se il costruttore di `PrintWriter` lancia un'eccezione?). Perché non si può risolvere il problema spostando la dichiarazione della variabile `out` all'interno del blocco `try`?

- \*\* **Esercizio R11.19.** Supponete che il programma visto nel Paragrafo 11.8 legga un file contenente questi valori:

0  
1  
2  
3

Quale sarà il risultato? Come si può migliorare il programma per fare in modo che fornisca una più accurata segnalazione degli errori?

- \*\* **Esercizio R11.20.** Il metodo `readFile` del Paragrafo 11.8 può lanciare un'eccezione di tipo `NullPointerException`? Se sì, in quale situazione?

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

# 12

## Ricorsione

### Obiettivi del capitolo

- Comprendere la tecnica della ricorsione
- Capire la relazione esistente tra ricorsione e iterazione
- Analizzare problemi che sono molto più semplici da risolvere con la ricorsione che con l'iterazione
- Imparare a “pensare ricorsivamente”
- Essere in grado di usare metodi ausiliari ricorsivi
- Capire come l'uso della ricorsione si ripercuote sull'efficienza di un algoritmo

La ricorsione è una potente tecnica per scomporre complessi problemi computazionali in problemi più semplici. 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 dovreste 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, 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 tale 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;
}
```

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'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: calcolate l'area del triangolo più piccolo, *senza pensare a come questo possa funzionare*, e aggiungete 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 gestiscono in modo diretto le elaborazioni più semplici.

**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 di 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 accade 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 veramente 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 i cicli equivalenti per molte ricorsioni complesse, come quella del nostro prossimo esempio, 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 ch12/triangle/Triangle.java

```
/**
 * Una forma triangolare composta di quadrati unitari impilati, come questa:
 * []
 * []
 * []
 * ...
 */
public class Triangle
{
    private int width;

    /**
     * Costruisce una forma triangolare.
```

```

@param aWidth l'ampiezza (e l'altezza) del triangolo
*/
public Triangle(int aWidth)
{
    width = aWidth;
}

/**
    Calcola l'area del triangolo.
    @return l'area
*/
public int getArea()
{
    if (width <= 0) { return 0; }
    if (width == 1) { return 1; }
    Triangle smallerTriangle = new Triangle(width - 1);
    int smallerArea = smallerTriangle.getArea();
    return smallerArea + width;
}
}

```

### File ch12/triangle/TriangleTester.java

```

public class TriangleTester
{
    public static void main(String[] args)
    {
        Triangle t = new Triangle(10);
        int area = t.getArea();
        System.out.println("Area: " + area);
        System.out.println("Expected: 55");
    }
}

```

### Esecuzione del programma

Area: 55  
Expected: 55

### Auto-valutazione

- Perché nel metodo `getArea` l'enunciato `if (width == 1) { return 1; }` non è necessario?
- Come modifichereste il programma per fare in modo che calcoli ricorsivamente l'area di un quadrato?

### Erri 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: il vostro programma termina bruscamente segnalando uno “stack overflow” (un errore di trabocco, *overflow*, nella pila, *stack*, che gestisce le invocazioni).

La ricorsione infinita accade o perché i valori dei parametri non diventano più semplici o 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 alquanto arduo. 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 fino al raggiungimento dell’enunciato `return smallerArea + width;`. 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?

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*, in alto a sinistra in 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 innestate ci si trova.



## Consigli pratici 12.1

### Pensare ricorsivamente

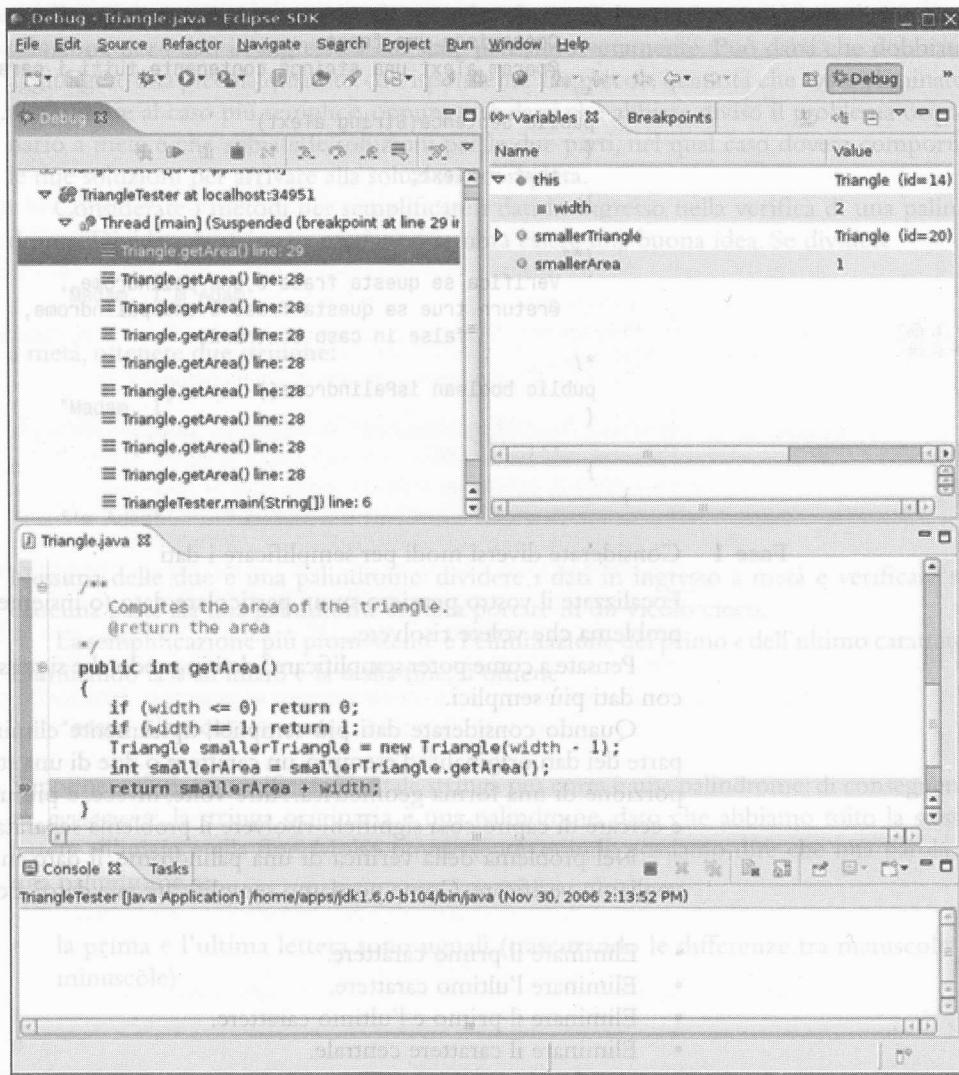
Risolvere un problema ricorsivamente richiede una diversa impostazione mentale rispetto alla soluzione realizzata mediante la programmazione di cicli. In effetti, la cosa risulta più facile se siete un po’ pigri (o fate finta di esserlo) e vi piace che altri facciano la maggior parte del vostro lavoro. 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 il meccanismo della ricorsione, consideriamo il seguente problema. Vogliamo verificare se una frase è una *palindrome* (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

**Figura 1**

Debugging di un metodo ricorsivo.



La parola che si ottiene eliminando la prima e l'ultima lettera è una palindroma e, ovviamente, la più antica palindroma fra tutte

- Madam, I'm Adam

Quando si deve verificare se una stringa è una palindroma, 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 Sentence
{
    private String text;

```

```

    /**
     * Costruisce una frase.
     * @param aText una stringa contenente tutti i caratteri di una frase
    */
    public Sentence(String aText)
    {
        text = aText;
    }

    /**
     * Verifica se questa frase è una palindrome.
     * @return true se questa frase è una palindrome,
             false in caso contrario
    */
    public boolean isPalindrome()
    {
        ...
    }
}

```

**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.

Quando considerate dati più semplici, tipicamente eliminate soltanto una piccola parte dei dati originali: ad esempio, un carattere o due 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 una palindrome, 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 una palindrome.

**Fase 2** Combinate le soluzioni dei casi più semplici per fornire una soluzione al 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 sapere o capire *come* si ottengano queste soluzioni, confidate semplicemente nella loro reale 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 in ingresso 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 una palindrome. 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 una palindrome: dividere i dati in ingresso a metà e verificare se ciascuna parte sia una palindrome 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 che tale stringa più corta è una palindrome: di conseguenza, ovviamente, la stringa originaria è una palindrome, dato che abbiamo tolto la stessa lettera all'inizio e alla fine. Molto promettente: quindi, possiamo dire che una parola è una palindrome 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 una palindrome.

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 un carattere *!*, che non è uguale al carattere *A* iniziale. Sappiamo, però, che quando verifichiamo se una stringa è una palindrome 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 una palindrome, allora rimane una palindrome 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 di ingresso, finché certamente giunge 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 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 dovrà essere attribuito al caso degenero 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 una palindrome:

- stringhe di due caratteri
- stringhe di 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, prenderci cura delle stringhe di lunghezza 0 e 1: in tali casi la Fase 2 non può essere applicata, perché non ci sono due caratteri da eliminare.

La stringa vuota è una palindrome: è identica a se stessa quando la leggete al contrario. Se pensate che questo ragionamento sia troppo artificioso, considerate la stringa "mm". Secondo la regola identificata nella Fase 2, questa stringa (che è certamente una palindrome) è una palindrome se il primo e l'ultimo carattere sono uguali e se la parte rimanente (cioè la stringa vuota) è una palindrome. Quindi, ha senso affermare che la stringa vuota è una palindrome.

Una stringa costituita da un'unica lettera, come "I", è una palindrome. 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 una palindrome. In definitiva, tutte le stringhe di lunghezza 0 e 1 sono palindromi.

**Fase 4** Scrivete il codice per la soluzione combinando i casi semplici e il passo di semplificazione

Ora siete pronti per scrivere la soluzione. Gestite separatamente i casi dei dati in ingresso speciali che avete considerato nella Fase 3. Se i dati non rientrano in uno di questi casi più semplici, seguite il ragionamento che avete identificato nella Fase 2.

Ecco il metodo `isPalindrome`.

```
public boolean isPalindrome()
{
    int length = text.length();

    // considera separatamente i casi delle stringhe più brevi
    if (length <= 1) { return true; }

    // 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
            Sentence shorter = new Sentence(text.substring(1, length - 1));
            return shorter.isPalindrome();
        }
        else
        {
            return false;
        }
    }
    else if (!Character.isLetter(last))
    {
        // elimina l'ultimo carattere
        Sentence shorter = new Sentence(text.substring(0, length - 1));
        return shorter.isPalindrome();
    }
    else
    {
        // elimina il primo carattere
        Sentence shorter = new Sentence(text.substring(1, length));
        return shorter.isPalindrome();
    }
}
```



## Esempi completi 12.1

### Cercare file

Dovete visualizzare il nome 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 di tipo `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 di tipo `File` contenuti in una cartella: possono essere semplici file o altre cartelle. Considerate come esempio questo albero di cartelle e ipotizzate che l'oggetto `f`, di tipo `File`, corrisponda alla cartella `bigjava`.



In questo caso, `f.isDirectory()` restituisce `true`, mentre `f.listFiles()` restituisce un array contenente gli oggetti di tipo `File` che rappresentano `bigjava/ch01` e `bigjava/ch02`.

#### Fase 1 Considerate diversi modi per semplificare i dati

Il nostro problema opera su due dati di ingresso: un oggetto di tipo `File`, che rappresenta un albero di cartelle, e un'estensione per nomi di file. È chiaro 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 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 fornire una soluzione al problema originario

Ci viene semplicemente chiesto di visualizzare i 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** Scrivete il codice per 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 root la radice dell'albero di cartelle
     */
    public FileFinder(File root)
    {
        children = root.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 `FileFinder.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 `ch12/file` del pacchetto di file scaricabili per questo libro completa la soluzione.

Abbiamo creato 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.

La soluzione completa si può trovare nel file `ch12/file/FileFinder2.java` del pacchetto di file scaricabili per questo libro.

```
import java.io.File;  
  
public class FileFinder2  
{  
    public static void main(String[] args)  
    {  
        File startingDirectory = new File("/home/myname");  
        find(startingDirectory, ".java");  
    }  
  
    /**  
     * 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);  
            }  
        }  
    }  
}
```

## 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 palindrome vista nei Consigli pratici 12.1: costruire nuovi oggetti di tipo `Sentence` a ogni passo è poco efficiente. Provate a considerare la seguente modifica al problema: invece di verificare se l'intera frase è una palindrome, verifichiamo se una sottostringa è una palindrome.

```
/**  
 * Verifica se una sottostringa della frase è una palindrome.  
 * @param start l'indice del primo carattere della sottostringa  
 * @param end l'indice dell'ultimo carattere della sottostringa  
 * @return true se la sottostringa è una palindrome  
 */  
public boolean isPalindrome(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 `Sentence` per rappresentare le stringhe che diventano sempre più brevi.

```
public boolean isPalindrome(int start, int end)  
{  
    // considera separatamente i casi delle stringhe di lunghezza 0 e 1  
    if (start >= end) { return true; }  
  
    // 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))  
    {  
        // entrambi i caratteri sono lettere  
        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 globale, perché l'utente del vostro metodo non deve sapere 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 boolean isPalindrome()
{
    return isPalindrome(0, text.length() - 1);
}

```

Notate che questo metodo *non* è ricorsivo: il metodo `isPalindrome()` invoca il metodo ausiliario, `isPalindrome(int, int)`. In questo esempio usiamo il sovraccarico per dichiarare due metodi aventi lo stesso nome: il metodo `isPalindrome` privo di parametri è quello destinato al pubblico utilizzo, mentre il secondo metodo, con 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 ogni qualvolta sia più semplice risolvere un problema ricorsivo leggermente diverso dal problema originale.

## Auto-valutazione

3. Dobbiamo necessariamente dare lo stesso nome ai due metodi che abbiamo chiamato `isPalindrome`?
4. In quale momento il metodo ricorsivo `isPalindrome` smette di invocare se stesso?

### 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 cattive prestazioni. 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 una funzione che calcola  $f_n$  per qualsiasi valore di  $n$ : traduciamo direttamente la definizione in un metodo ricorsivo.

### File ch12/fib/RecursiveFib.java

```
import java.util.Scanner;

/**
 * Questo programma calcola i numeri di Fibonacci
 * usando un metodo ricorsivo.
 */
public class RecursiveFib
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter n: ");
        int n = in.nextInt();

        for (int i = 1; i <= n; i++)
        {
            long f = fib(i);
            System.out.println("fib(" + i + ") = " + f);
        }
    }

    /**
     * Calcola un numero di Fibonacci.
     * @param n un numero intero
     * @return l'n-esimo numero di Fibonacci
     */
    public static long fib(int n)
    {
        if (n <= 2) { return 1; }
        else return fib(n - 1) + fib(n - 2);
    }
}
```

### 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*.

### File ch12/fib/RecursiveFibTracer.java

```
import java.util.Scanner;

/**
 * Questo programma stampa messaggi di tracciatura
 * che mostrano quanto spesso, per calcolare i numeri
 * di Fibonacci, il metodo ricorsivo invoca se stesso.
 */
public class RecursiveFibTracer
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter n: ");
        int n = in.nextInt();

        long f = fib(n);

        System.out.println("fib(" + n + ") = " + f);
    }

    /**
     * Calcola un numero di Fibonacci.
     * @param n un numero intero
     * @return l'n-esimo numero di Fibonacci
     */
    public static long fib(int n)
    {
        System.out.println("Entering fib: n = " + n);
        long f;
        if (n <= 2) { f = 1; }
        else { f = fib(n - 1) + fib(n - 2); }
        System.out.println("Exiting fib: n = " + n
                           + " return value = " + f);
        return f;
    }
}
```

### 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

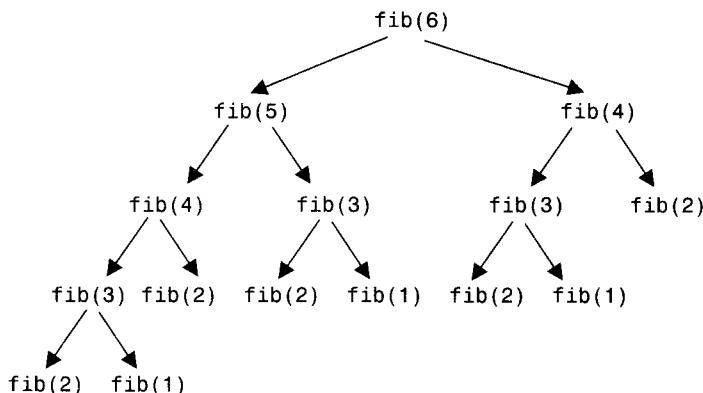
```

La Figura 2 mostra l'albero 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 anoteremmo 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.

**Figura 2**

Schema delle invocazioni del metodo ricorsivo `fib`



### File ch12/fib/LoopFib.java

```

import java.util.Scanner;

/**
 * Questo programma calcola i numeri di Fibonacci
 * usando un metodo iterativo.
 */
public class LoopFib
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter n: ");
        int n = in.nextInt();

        for (int i = 1; i <= n; i++)
        {
            long f = fib(i);
            System.out.println("fib(" + i + ") = " + f);
        }
    }

    /**
     * Calcola un numero di Fibonacci.
     * @param n un numero intero
     * @return l'n-esimo numero di Fibonacci
     */
    public static long fib(int n)
    {
        if (n <= 2) { return 1; }
        long olderValue = 1;
        long oldValue = 1;
        long newValue = 1;
        for (int i = 3; i <= n; i++)
        {
            newValue = oldValue + olderValue;
            olderValue = oldValue;
            oldValue = newValue;
        }
        return newValue;
    }
}

```

### 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

```

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.

Questo metodo viene eseguito *molto* più velocemente della versione ricorsiva.

In questo esempio relativo al metodo `fib` la soluzione ricorsiva era più facile da realizzare perché seguiva 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 una palindroma.

```
public boolean isPalindrome()
{
    int start = 0;
    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 una palindroma 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, poiché 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.

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".

In molti casi una soluzione ricorsiva  
è più facile da capire e da realizzare  
correttamente, rispetto  
a una soluzione iterativa.



## Auto-valutazione

5. È più veloce calcolare i numeri triangolari ricorsivamente, come visto nel Paragrafo 12.1, oppure usando un ciclo che calcoli  $1 + 2 + 3 + \dots + \text{width}$ ?
6. 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?

## 12.4 Permutazioni

Le permutazioni di una stringa  
si possono ottenere in modo più  
naturale tramite 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 P12.11, 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"
```

Come nel paragrafo precedente, dichiareremo una classe avente il compito di calcolare la risposta, che, in questo caso, non è semplicemente un numero, ma un insieme di stringhe permutate. Ecco la classe che le genera:

```
public class PermutationGenerator
{
    public PermutationGenerator(String aWord) {...}
    public ArrayList<String> getPermutations() {...}
}
```

Ecco il programma di prova che visualizza tutte le permutazioni della stringa "eat".

### File ch12/permute/PermutationGeneratorDemo.java

```

import java.util.ArrayList;

/**
 * Questo programma utilizza il generatore di permutazioni.
 */
public class PermutationGeneratorDemo
{
    public static void main(String[] args)
    {
        PermutationGenerator generator
            = new PermutationGenerator("eat");
        ArrayList<String> permutations = generator.getPermutations();
        for (String s : permutations)
        {
            System.out.println(s);
        }
    }
}

```

### Esecuzione del programma

```

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, la realizzazione della classe è quasi banale. Nel metodo `getPermutations`, 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 shorterWord = word.substring(0, i) + word.substring(i + 1);
```

Costruiamo, poi, un generatore di permutazioni che fornisca le permutazioni di tale parola più breve:

```
PermutationGenerator shorterPermutationGenerator
    = new PermutationGenerator(shorterWord);
ArrayList<String> shorterWordPermutations
    = shorterPermutationGenerator.getPermutations();
```

Infine, aggiungiamo a tutte le permutazioni della parola più breve il carattere precedentemente escluso:

```
for (String s : shorterWordPermutations)
{
    permutations.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 `PermutationGenerator` completa.

### File ch12/permute/PermutationGenerator.java

```
import java.util.ArrayList;

/**
 * Questa classe genera le permutazioni di una parola.
 */
public class PermutationGenerator
{
    private String word;

    /**
     * Costruisce un generatore di permutazioni.
     * @param aWord la parola da permutare
     */
    public PermutationGenerator(String aWord)
    {
        word = aWord;
    }
```

```

    /**
     * Fornisce tutte le permutazioni della parola.
     * @return un vettore contenente tutte le permutazioni
    */
public ArrayList<String> getPermutations()
{
    ArrayList<String> permutations = new ArrayList<String>();

    // la stringa vuota ha un'unica permutazione: se stessa
    if (word.length() == 0)
    {
        permutations.add(word);
        return permutations;
    }

    // effettua un ciclo esaminando tutti i caratteri della stringa
    for (int i = 0; i < word.length(); i++)
    {
        // componi una parola più breve eliminando
        // il carattere i-esimo
        String shorterWord = word.substring(0, i)
            + word.substring(i + 1);

        // genera tutte le permutazioni della parola più breve
        PermutationGenerator shorterPermutationGenerator
            = new PermutationGenerator(shorterWord);
        ArrayList<String> shorterWordPermutations
            = shorterPermutationGenerator.getPermutations();

        // aggiungi il carattere escluso all'inizio di ciascuna
        // permutazione della parola più breve
        for (String s : shorterWordPermutations)
        {
            permutations.add(word.charAt(i) + s);
        }
    }
    // restituisce tutte le permutazioni
    return permutations;
}
}

```

Confrontate le classi `PermutationGenerator` e `Triangle`, che funzionano secondo lo stesso principio: quando elaborano dati complessi in ingresso, generano un altro oggetto della stessa classe che elabora dati più semplici; quindi, combinano il lavoro svolto da quell'oggetto con il proprio, per fornire i risultati per i dati più complessi. Non c'è davvero alcuna particolare complessità in questa procedura, se pensate alla soluzione a questo livello: dietro le quinte, però, l'oggetto che deve elaborare i dati più semplici crea, a sua volta, un altro oggetto che elabora dati ancora più semplici, il quale crea ancora un altro oggetto, e così via, finché i dati che devono essere elaborati sono tanto 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

7. Quali sono le permutazioni della parola di quattro lettere **beat**?
8. 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?
9. Perché non è facile sviluppare una soluzione iterativa per la generazione di permutazioni?

## 12.5 Ricorsione mutua

In una ricorsione mutua, un insieme di metodi cooperanti 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 è 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 sottoespressioni.

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  $3+4*5$ . Entrando nel diagramma sintattico corrispondente a una *espressione* (*expression*), la freccia punta direttamente a un *termine* (*term*), per cui non abbiamo alternative: dobbiamo entrare nel diagramma che descrive un termine, dove la freccia punta a un *fattore* (*factor*), lasciandoci nuovamente senza alternative. Entra 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: accettiamo il dato in ingresso perché corrisponde alla definizione di *numero* (*number*). Seguiamo la freccia uscente da *numero* fino alla fine di *fattore*: proprio come avviene in un'invocazione di 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 ricevuto 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*.

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



## Note di cronaca 12.1

### Ilimiti 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 svilupperlo, per venderlo alle università di tutto il mondo.

Tuttavia, prima che iniziiate a raccolgere capitoli per questa impresa, 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

#### Una 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                          | 1               | sinistra                    | 4                                      |
| 2                    | 0                          | 0               | destra                      | 2                                      |
| 2                    | 1                          | 1               | destra                      | 2                                      |
| 2                    | 2                          | 0               | sinistra                    | 3                                      |
| 3                    | 0                          | 0               | sinistra                    | 3                                      |
| 3                    | 1                          | 1               | sinistra                    | 3                                      |
| 3                    | 2                          | 2               | destra                      | 1                                      |
| 4                    | 1                          | 1               | destra                      | 5                                      |
| 4                    | 2                          | 0               | sinistra                    | 4                                      |

##### Nastro

#### Unità di controllo

Testina lettura/scrittura

eseguito con  $I$  come dato di ingresso, terminerà senza entrare in un ciclo infinito". Ovviamente, per alcuni tipi di programmi e di dati in ingresso è 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 dovreste 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[] args)
    {
        String r = legge i dati in
            ingresso del programma;
        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, un attimo: ciò non può essere esatto. Ciò significherebbe

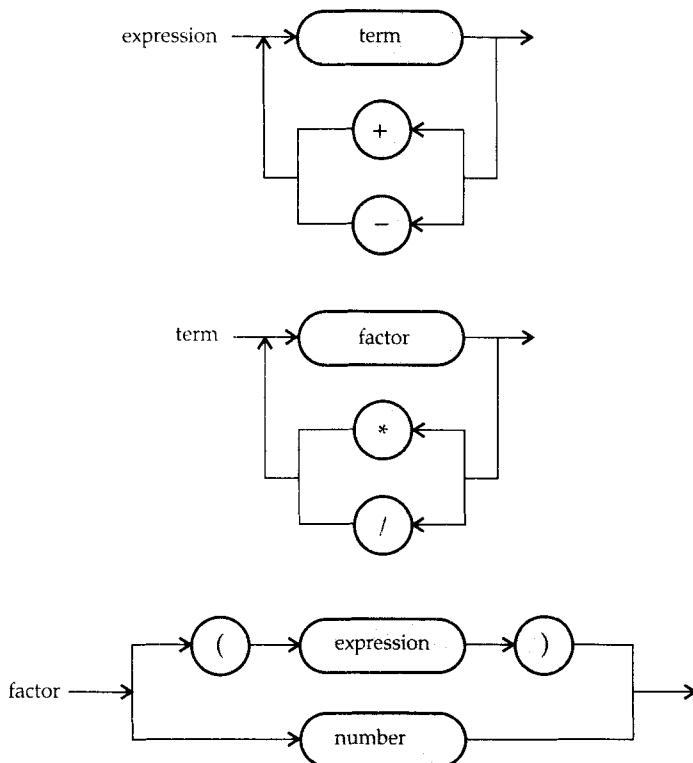
che `checker.check(r, r)` restituisce `false` quando  $r$  è il codice sorgente di  $K$ . Come potete facilmente vedere, in tal caso il metodo `main` del programma killer termina, per cui  $K$  non è entrato in un ciclo infinito. Ciò mostra che  $K$  deve terminare quando analizza se stesso, per cui `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 *qualsiasi* programma 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 ad 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.

**Figura 3**

Diagrammi sintattici per la valutazione di un'espressione



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  $(3+4)*5$ .

Per quale motivo 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.

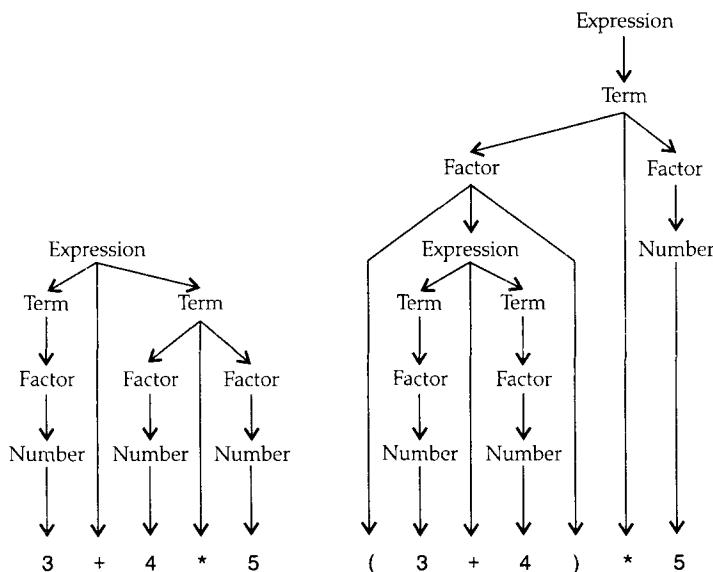
Alla fine di questo paragrafo troverete l'implementazione della classe `Evaluator`, che valuta queste espressioni. Un oggetto di tipo `Evaluator` usa la classe `ExpressionTokenizer`, che scomponete una stringa in elementi (*tokens*): numeri, operatori e parentesi (per semplicità, accettiamo come numeri 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 è 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 considerato successivamente.

Per calcolare il valore di un'espressione, realizziamo tre metodi: `getExpressionValue`, `getTermValue` e `getFactorValue`. Dapprima il metodo `getExpressionValue` invoca

**Figura 4**

Alberi sintattici per due espressioni



`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 assicurarcì 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 ad ogni invocazione ricorsiva alcuni elementi della stringa di ingresso vengono estratti, la ricorsione deve necessariamente terminare.

### File ch12/expr/Evaluator.java

```
/*
 * Una classe che può calcolare il valore di una
 * espressione aritmetica.
 */
public class Evaluator
{
    private ExpressionTokenizer tokenizer;

    /**
     * Costruisce un valutatore.
     */
```

```
    @param anExpression una stringa che contiene
                      l'espressione da valutare
*/
public Evaluator(String anExpression)
{
    tokenizer = new ExpressionTokenizer(anExpression);
}

/**
    Valuta l'espressione.
    @return il valore dell'espressione
*/
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;
}

/**
    Valuta il successivo termine nell'espressione.
    @return il valore del termine
*/
public int getTermValue()
{
    int value = getFactorValue();
    boolean done = false;
    while (!done)
    {
        String next = tokenizer.peekToken();
        if ("*".equals(next) || "/".equals(next))
        {
            tokenizer.nextToken();
            int value2 = getFactorValue();
            if ("*".equals(next)) { value = value * value2; }
            else { value = value / value2; }
        }
        else
        {
            done = true;
        }
    }
}
```

```

        }
        return value;
    }

    /**
     * Valuta il successivo fattore nell'espressione.
     * @return il valore del fattore
    */
    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;
    }
}

```

### File ch12/expr/ExpressionTokenizer.java

```

    /**
     * Questa classe scomponete una stringa che descrive
     * un'espressione in elementi (o token):
     * numeri, parentesi tonde e operatori.
    */
    public class ExpressionTokenizer
    {
        private String input;
        private int start; // l'inizio del token attuale
        private int end; // la posizione successiva all'ultima del token attuale

        /**
         * Costruisce uno scompositore.
         * @param anInput la stringa da scomporre
        */
        public ExpressionTokenizer(String anInput)
        {
            input = anInput;
            start = 0;
            end = 0;
            nextToken(); // cerca il primo token
        }

        /**
         * Restituisce il token successivo senza estrarre.
         * @return il token successivo, oppure null se
         *         non ci sono più token
        */

```

```

        */
    public String peekToken()
    {
        if (start >= input.length()) { return null; }
        else { return input.substring(start, end); }
    }

    /**
     Restituisce il token successivo e fa avanzare
     lo scompositore al token seguente.
     @return il token successivo, oppure null se
             non ci sono più token
    */
    public String nextToken()
    {
        String r = peekToken();
        start = end;
        if (start >= input.length()) { return r; }
        if (Character.isDigit(input.charAt(start)))
        {
            end = start + 1;
            while (end < input.length()
                   && Character.isDigit(input.charAt(end)))
            {
                end++;
            }
        }
        else
        {
            end = start + 1;
        }
        return r;
    }
}

```

### File ch12/expr/ExpressionCalculator.java

```

import java.util.Scanner;

/**
 * Questo programma calcola il valore di un'espressione
 * costituita da numeri, operatori aritmetici e parentesi tonde.
 */
public class ExpressionCalculator
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter an expression: ");
        String input = in.nextLine();
        Evaluator e = new Evaluator(input);
        int value = e.getExpressionValue();
        System.out.println(input + " = " + value);
    }
}

```

## Esecuzione del programma

```
Enter an expression: 3+4*5
3+4*5=23
```



## Auto-valutazione

10. Qual è la differenza tra termine e fattore? Perché ci servono entrambi questi concetti?
11. Perché l'analizzatore sintattico di espressioni (“parser”) usa la ricorsione mutua?
12. Cosa succede se cercate di analizzare sintatticamente l'espressione  $3+4^*)5$ , che non è valida? In particolare, quale metodo lancia un'eccezione?

## 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 di 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.

### Le ricorsioni più complesse non si possono risolvere con un semplice ciclo

- Le permutazioni di una stringa si possono ottenere in modo più naturale tramite ricorsione piuttosto che usando un ciclo.

### La ricorsione mutua: esempio di un analizzatore sintattico

- In una ricorsione mutua, un insieme di metodi cooperanti si invocano l'un l'altro ripetutamente.

## Esercizi di ripasso

- \* **Esercizio R12.1.** Definite i seguenti termini:

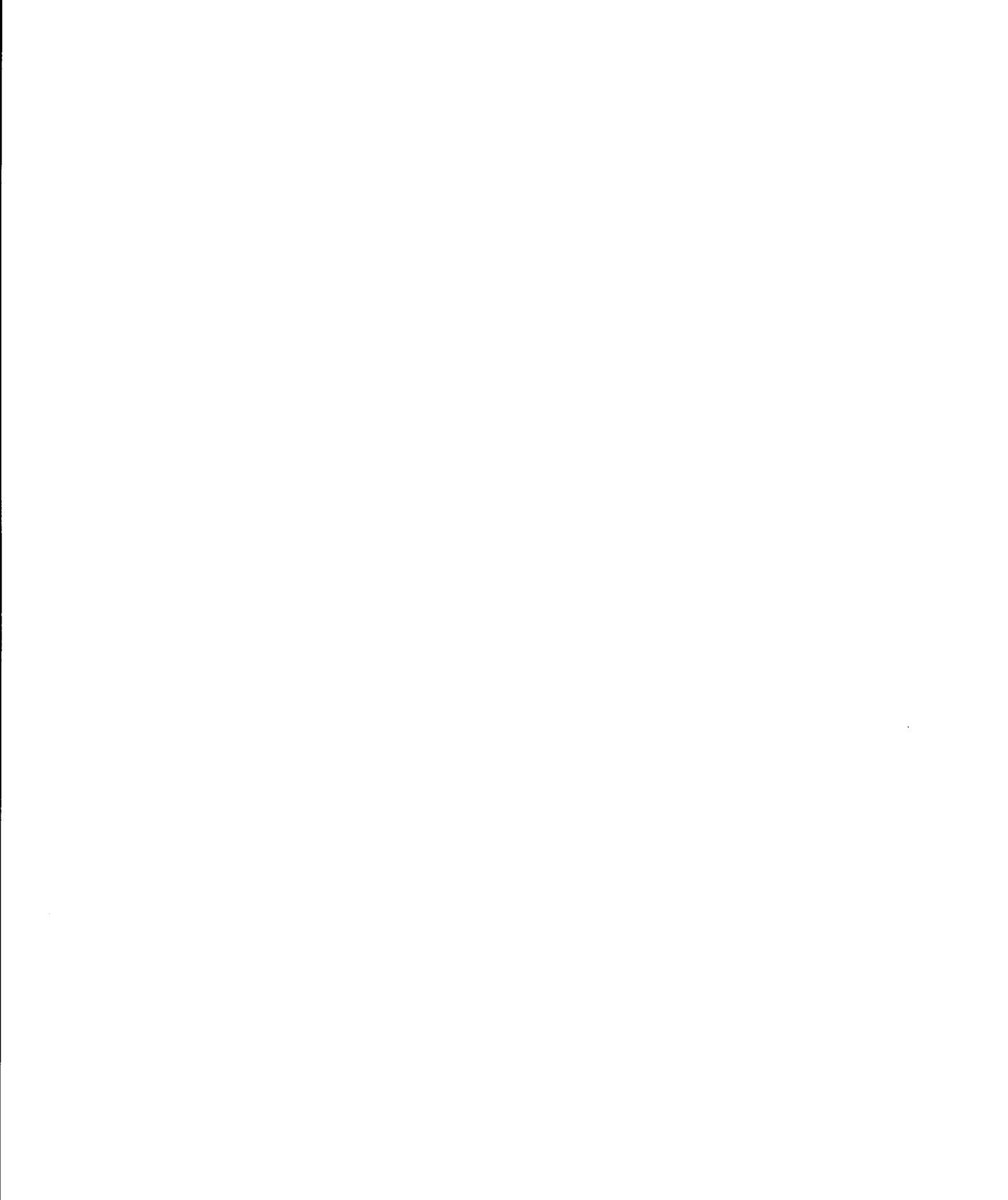
- a. ricorsione
- b. iterazione
- c. ricorsione infinita
- d. metodo ausiliario ricorsivo

- ★★ **Esercizio R12.2.** Delineate, senza implementarla, una soluzione ricorsiva per trovare il valore minore in un array.
- ★★ **Esercizio R12.3.** Delineate, senza implementarla, una soluzione ricorsiva per ordinare un array di numeri. *Suggerimento:* per prima cosa trovate il valore minore nell'array.
- ★★ **Esercizio R12.4.** Delineate, senza implementarla, una soluzione ricorsiva per generare tutti i sottoinsiemi dell'insieme  $\{1, 2, \dots, n\}$
- ★★★ **Esercizio R12.5.** L'Esercizio P12.12 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.
- \* **Esercizio 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?
- ★★ **Esercizio 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.
- \* **Esercizio 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.
- ★★ **Esercizio R12.9.** Scoprite quanto spesso la versione ricorsiva di fib invoca se stessa. Usate una variabile statica, `fibCount`, e incrementatela ad ogni invocazione di fib. Qual è la relazione tra `fib(n)` e `fibCount`?
- ★★★ **Esercizio R12.10.** Quante mosse sono necessarie nel problema della “Torre di Hanoi” dell'Esercizio P12.13 per spostare  $n$  dischi? *Suggerimento:* come spiegato nell'esercizio:

$$\text{mosse}(1) = 1$$

$$\text{mosse}(n) = 2 \cdot \text{mosse}(n - 1) + 1$$

Sul sito web dedicato al libro si trova una vasta raccolta di esercizi di programmazione e di progetti 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
- Imparare a misurare il tempo d'esecuzione di un programma

L'ordinamento e la ricerca sono tra le operazioni più frequenti nell'elaborazione di dati e, ovviamente, la libreria di Java offre metodi per eseguirle; nonostante ciò, è davvero utile studiare algoritmi per risolvere questi problemi. Dovete imparare ad analizzare le prestazioni di algoritmi e a scegliere quello migliore per un determinato compito: farlo in relazione a ordinamento e ricerca è un ottimo punto di partenza, perché si tratta di problemi facili da capire. Come vedrete in questo capitolo, gli algoritmi più semplici non hanno buone prestazioni: usando algoritmi più sofisticati possiamo ottenere miglioramenti rilevanti.

## 13.1 Ordinamento per selezione

In questo paragrafo illustreremo un algoritmo di ordinamento, il primo dei pochi che vedremo. Un *algoritmo di ordinamento* (“sorting algorithm”) sposta 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 minore della regione terminale non ancora ordinata, spostandolo all’inizio della regione 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 memorizzato, precisamente 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 sicurezza 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 precedente, la zona omobreggiata indica la porzione dell’array che è già stata ordinata, mentre la parte rimanente è ancora da ordinare.

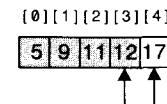
Successivamente, cerchiamo l’elemento minimo tra quelli rimanenti,  $a[1] \dots a[4]$ . Tale valore minimo, 9, si trova già nella posizione giusta: in questo caso non dobbiamo fare nulla e possiamo semplicemente estendere di una posizione verso destra la porzione dell’array che risulta essere già ordinata.

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] |
| 5   | 9   | 17  | 11  | 12  |

Ripetiamo il procedimento. Il valore minimo nella regione non ordinata è 11, che deve essere scambiato col primo valore di tale regione, 17.

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] |
| 5   | 9   | 17  | 11  | 12  |

Ora la regione 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.



Questo ci porta ad avere una regione non ancora elaborata di lunghezza 1, ma, naturalmente, una regione di lunghezza 1 è sempre ordinata. Abbiamo finito.

Proviamo a scrivere il codice per questo algoritmo. Per questo programma, come pure per gli altri programmi di questo capitolo, utilizzeremo un metodo ausiliario per generare un array riempito di valori casuali: lo inseriamo in una 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`.

Questo algoritmo ordinerà un array di numeri interi. Se la velocità non fosse un problema o se, semplicemente, non ci fossero a disposizione 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.

In Argomenti avanzati 13.1 viene presentato l'algoritmo di ordinamento per inserimento (“insertion sort”), un altro algoritmo di ordinamento molto semplice.

### File ch13/selsort/SelectionSorter.java

```
/*
 * Questa classe ordina un array
 * usando l'algoritmo di ordinamento per selezione.
 */
public class SelectionSorter
{
    private int[] a;

    /**
     * Costruisce un ordinatore per selezione.
     * @param anArray l'array da ordinare
     */
    public SelectionSorter(int[] anArray)
    {
        a = anArray;
    }

    /**
     * Ordina l'array gestito da questo ordinatore.
     */
    public void sort()
    {
        for (int i = 0; i < a.length - 1; i++)
        {
            int minPos = minimumPosition(i);
            swap(minPos, i);
        }
    }
}
```

```

    Trova l'elemento minimo in una parte terminale dell'array.
    @param from la prima posizione da considerare nell'array a
    @return la posizione dell'elemento minimo presente
            nell'intervallo a[from]...a[a.length - 1]
*/
private int minimumPosition(int from)
{
    int minPos = from;
    for (int i = from + 1; i < a.length; i++)
        if (a[i] < a[minPos]) minPos = i;
    return minPos;
}

/**
    Scambia due elementi nell'array.
    @param i la posizione del primo elemento
    @param j la posizione del secondo elemento
*/
private void swap(int i, int j)
{
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
}

```

### File ch13/selsort/SelectionSortDemo.java

```

import java.util.Arrays;

/**
    Questo programma applica l'algoritmo di ordinamento
    per selezione a un array riempito con numeri casuali.
*/
public class SelectionSortDemo
{
    public static void main(String[] args)
    {
        int[] a = ArrayUtil.randomIntArray(20, 100);
        System.out.println(Arrays.toString(a));

        SelectionSorter sorter = new SelectionSorter(a);
        sorter.sort();

        System.out.println(Arrays.toString(a));
    }
}

```

### File ch13/selsort/ArrayUtil.java

```

import java.util.Random;

/**
    Questa classe contiene metodi utili per
    la manipolazione di array.
*/

```

```

public class ArrayUtil
{
    private static Random generator = new Random();

    /**
     * Costruisce un array contenente numeri interi casuali.
     * @param length la lunghezza dell'array
     * @param n il numero di valori diversi possibili
     * @return un array contenente length numeri
     *         casuali compresi fra 0 e n - 1
    */
    public static int[] randomIntArray(int length, int n)
    {
        int[] a = new int[length];
        for (int i = 0; i < a.length; i++)
            a[i] = generator.nextInt(n);

        return a;
    }
}

```

### 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 se assegnassimo 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?

## 13.2 Misurazione delle prestazioni dell'ordinamento per selezione

Per rilevare 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 un programma, per la sua esecuzione, impiega una quantità di tempo percepibile, una certa quantità di quel tempo viene semplicemente usata per caricare il programma dal disco alla memoria o per visualizzare i risultati sullo schermo (cose per le quali non dovremmo penalizzarlo).

Per misurare in modo più accurato il tempo di esecuzione di un algoritmo, progettiamo una 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. Non ci interessa il numero assoluto di secondi trascorsi da quell'istante particolare, ovviamente, 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 ch13/selSort/StopWatch.java

```
/**  
 * Un cronometro misura il tempo che trascorre mentre è in azione.  
 * Potete avviare e arrestare ripetutamente il cronometro.  
 * Potete utilizzare un cronometro per misurare il tempo  
 * di esecuzione di un programma.  
 */  
public class StopWatch  
{  
    private long elapsedTime;  
    private long startTime;  
    private boolean isRunning;  
  
    /**  
     * Costruisce un cronometro fermo e azzerato.  
     */  
    public StopWatch()  
    {  
        reset();  
    }  
  
    /**  
     * Fa partire il cronometro, iniziando a misurare il tempo.  
     */  
    public void start()  
    {  
        if (isRunning) return;  
        isRunning = true;  
        startTime = System.currentTimeMillis();  
    }  
  
    /**  
     * Ferma il cronometro. Il tempo non viene più misurato  
     * e il tempo trascorso dall'ultimo avvio del cronometro  
     * viene sommato al tempo totale misurato.  
     */  
    public void stop()  
    {  
        if (!isRunning) return;  
        isRunning = false;  
        long endTime = System.currentTimeMillis();  
        elapsedTime = elapsedTime + endTime - startTime;  
    }  
  
    /**  
     * Restituisce il tempo totale misurato.  
     * @return il tempo totale misurato  
     */  
    public long getElapsedTime()  
    {  
        if (isRunning)  
        {
```

```
        long endTime = System.currentTimeMillis();
        return = elapsedTime + endTime - startTime;
    }
    else
        return elapsedTime;
}

/**
 * Ferma il cronometro e azzera il tempo totale.
 */
public void reset()
{
    elapsedTime = 0;
    isRunning = false;
}
}
```

Ed ecco come utilizzeremo il cronometro per misurare le prestazioni dell'algoritmo di ordinamento.

### File ch13/selsort/SelectionSortTimer.java

```
import java.util.Scanner;

/**
 * Questo programma misura il tempo richiesto per
 * ordinare, con l'algoritmo di ordinamento per selezione,
 * un array di dimensione specificata dall'utente.
 */
public class SelectionSortTimer
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter array size: ");
        int n = in.nextInt();

        // costruisce un array casuale

        int[] a = ArrayUtil.randomIntArray(n, 100);
        SelectionSorter sorter = new SelectionSorter(a);

        // usa il cronometro per misurare il tempo

        StopWatch timer = new StopWatch();

        timer.start();
        sorter.sort(a);
        timer.stop();

        System.out.println("Elapsed time: "
                           + timer.getElapsedTime() + " milliseconds");
    }
}
```

### Esempio di esecuzione del programma

```
Enter array size: 100000
Elapsed time: 27880 milliseconds
```

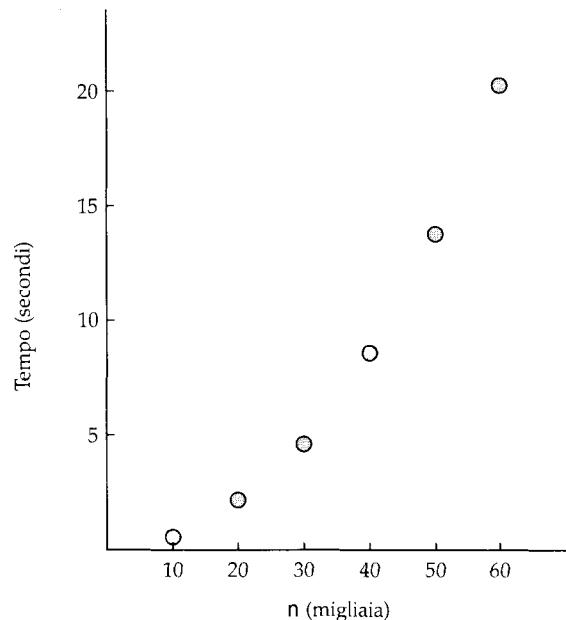
Per misurare il tempo di esecuzione di un metodo, chiedete l'ora al sistema subito prima e subito dopo l'invocazione del metodo.

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 lettura dei dati forniti dall'utente e per la visualizzazione dei risultati.

La tabella di Figura 1 mostra i risultati di alcune esecuzioni del programma. Queste rilevazioni sono state ottenute con un processore Intel operante 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 più del doppio.

**Figura 1**  
Tempo impiegato dall'ordinamento per selezione.



| n      | Millisecondi |
|--------|--------------|
| 10 000 | 786          |
| 20 000 | 2148         |
| 30 000 | 4796         |
| 40 000 | 9192         |
| 50 000 | 13 321       |
| 60 000 | 19 299       |



### Auto-valutazione

3. Quanti secondi sarebbero necessari, approssimativamente, per ordinare un insieme di dati contenente 80 000 valori?
4. Osservate il grafico della Figura 1: a quale curva matematica assomiglia?

### 13.3 Analisi delle prestazioni dell'algoritmo di ordinamento per selezione

Proviamo a conteggiare 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 conteggiare il numero di *visite* di elementi dell'array. Ciascuna visita richiede all'incirca la stessa quantità di lavoro di altre operazioni, quali l'incremento di indici o il confronto di valori.

Supponiamo che  $n$  sia la dimensione dell'array. Per prima cosa, dobbiamo trovare il più piccolo 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 trovare il minimo. Ciascun passo richiede 2 visite per scambiare gli elementi. Quindi, il numero totale delle visite è:

$$\begin{aligned} n + 2 + (n - 1) + 2 + \dots + 2 + 2 &= n + (n - 1) + \dots + 2 + (n - 1) \cdot 2 \\ &= 2 + \dots + (n - 1) + n + (n - 1) \cdot 2 \\ &= \frac{n \cdot (n + 1)}{2} - 1 + (n - 1) \cdot 2 \end{aligned}$$

perché

$$1 + 2 + \dots + (n - 1) + n = \frac{n \cdot (n + 1)}{2}$$

Sviluppando le moltiplicazioni e raccogliendo  $n$  a fattore comune, troviamo che il numero delle visite è:

$$\frac{1}{2}n^2 + \frac{5}{2}n - 3$$

Ottieniamo 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,  $(1/2)n^2$  è 500 000 o 2 000 000. Il termine di grado inferiore,  $(5/2)n - 3$ , non contribuisce più di tanto, vale appena 2497 oppure 4997, una goccia nel mare rispetto alle centinaia di migliaia o addirittura ai milioni di visite corrispondenti al termine  $(1/2)n^2$ . Semplicemente, ignoreremo questi termini di grado inferiore, così come il fattore costante 1/2: non ci interessa il conteggio effettivo delle visite per un singolo valore di  $n$ , vogliamo soltanto confrontare i rapporti dei conteggi 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 vedere che il numero di visite quadruplica quando le dimensioni dell’array raddoppiano:  $(2n)^2 = 4n^2$ .

Per indicare che il numero delle visite è dell’ordine di  $n^2$ , gli informatici teorici spesso usano la *notazione O-grande* (*big-Oh*, in inglese). Il numero delle visite è  $O(n^2)$ : si tratta di una comoda abbreviazione.

Gli informatici teorici usano la notazione O-grande: se  $f(n) = O(g(n))$ , la funzione  $f$  non aumenta più rapidamente della funzione  $g$ .

In generale, l’espressione  $f(n) = O(g(n))$  significa che  $f$  non cresce più rapidamente di  $g$ , oppure, in modo più formale: per tutti i valori di  $n$  maggiori di una certa soglia, si ha  $f(n)/g(n) \leq C$  per un qualche valore costante  $C$ . Solitamente la funzione  $g$  viene scelta molto semplice, come  $n^2$  nel nostro esempio.

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 a un elemento 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  $O(n^2)$ .

L’ordinamento per selezione è un algoritmo  $O(n^2)$ : il raddoppio della dimensione dell’insieme di dati quadruplica il tempo di elaborazione.

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 10 000. Per ordinare un array con un milione di voci (per generare, ad esempio, un elenco telefonico), si impiega un tempo 10 000 volte più lungo di quello che occorrerebbe per ordinare 10 000 voci. Se 10 000 voci si possono ordinare in circa mezzo secondo (come nel nostro esempio), allora l’ordinamento di un milione di voci richiede ben più di un’ora. 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.



## Auto-valutazione

5. 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?
6. Quanto deve valere  $n$  perché  $(1/2)n^2$  sia maggiore di  $(5/2)n - 3$ ?

## Argomenti avanzati 13.1

### Ordinamento per inserimento

L'ordinamento per inserimento 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 |
|---|----|----|---|---|

Successivamente, aggiungiamo l'elemento  $a[2]$ , il cui valore è 16: casualmente, non c'è bisogno di spostare tale elemento.

|   |    |    |   |   |
|---|----|----|---|---|
| 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
{
    private int[] a;

    /**
     * Costruisce un ordinatore per inserimento.
     * @param anArray l'array da ordinare
     */
    public InsertionSorter(int[] anArray)
    {
        a = anArray;
    }
}
```

```

    /**
     * Ordina l'array gestito da questo ordinatore.
    */
public void sort()
{
    for (int i = 1; i < a.length; i++)
    {
        int next = a[i];
        // cerca la posizione in cui inserire, spostando
        // in posizioni di indice superiore tutti gli elementi
        // di valore maggiore
        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 R13.3). Tale proprietà è utile in molti casi pratici, dal momento che spesso capita di dover ordinare insiemi di dati già parzialmente ordinati.



## Argomenti avanzati 13.2

### $\Omega$ -grande, $\Omega$ ( $\Omega$ ) e $\Theta$ ( $\Theta$ )

In questo capitolo abbiamo usato in modo un po' approssimativo la notazione  $O$ -grande, per descrivere il modo in cui cresce una funzione. Parlando in modo più corretto, l'espressione  $f(n) = O(g(n))$  significa che  $f$  non cresce più velocemente di  $g$ , ma è possibile che  $f$  cresca molto più lentamente, per cui tecnicamente è corretto dire che  $f(n) = n^2 + 5n - 3$  è  $O(n^3)$  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

$$f(n) = \Omega(g(n))$$

significa che  $f$  cresce almeno tanto velocemente quanto cresce  $g$  oppure, in modo più formale: per tutti i valori di  $n$  maggiori di una certa soglia, si ha  $f(n)/g(n) \geq C$  per un qualche valore costante  $C$  (il simbolo  $\Omega$  è la lettera omega maiuscola dell'alfabeto greco). Ad esempio,  $f(n) = n^2 + 5n - 3$  è  $\Omega(n^2)$  o anche  $\Omega(n)$ . L'espressione

$$f(n) = \Theta(g(n))$$

significa che  $f$  e  $g$  crescono con la stessa velocità, cioè è vero sia che  $f(n) = O(g(n))$  sia che  $f(n) = \Omega(g(n))$  (il simbolo  $\Theta$  è la lettera theta maiuscola dell'alfabeto greco).

La notazione  $\Theta$  fornisce la più precisa descrizione dell'andamento della crescita di una funzione. Ad esempio,  $f(n) = n^2 + 5n - 3$  è  $\Theta(n^2)$  ma non è  $\Theta(n)$  né  $\Theta(n^3)$ .

Le notazioni  $\Theta$  e  $\Omega$  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.

## 13.4 Ordinamento per fusione (MergeSort)

In questo paragrafo imparerete 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 sottoarray, scegliendo ogni volta l'elemento più piccolo:

|   |   |    |    |    |   |   |    |    |    |   |   |   |   |    |    |    |    |    |    |
|---|---|----|----|----|---|---|----|----|----|---|---|---|---|----|----|----|----|----|----|
| 5 | 9 | 10 | 12 | 17 | + | 8 | 11 | 20 | 32 | 1 |   |   |   |    |    |    |    |    |    |
| 5 | 9 | 10 | 12 | 17 | + | 8 | 11 | 20 | 32 | 1 | 5 |   |   |    |    |    |    |    |    |
| 5 | 9 | 10 | 12 | 17 | + | 8 | 11 | 20 | 32 | 1 | 5 | 8 |   |    |    |    |    |    |    |
| 5 | 9 | 10 | 12 | 17 | + | 8 | 11 | 20 | 32 | 1 | 5 | 8 | 9 |    |    |    |    |    |    |
| 5 | 9 | 10 | 12 | 17 | + | 8 | 11 | 20 | 32 | 1 | 5 | 8 | 9 | 10 |    |    |    |    |    |
| 5 | 9 | 10 | 12 | 17 | + | 8 | 11 | 20 | 32 | 1 | 5 | 8 | 9 | 10 | 11 |    |    |    |    |
| 5 | 9 | 10 | 12 | 17 | + | 8 | 11 | 20 | 32 | 1 | 5 | 8 | 9 | 10 | 11 | 12 |    |    |    |
| 5 | 9 | 10 | 12 | 17 | + | 8 | 11 | 20 | 32 | 1 | 5 | 8 | 9 | 10 | 11 | 12 | 17 |    |    |
| 5 | 9 | 10 | 12 | 17 | + | 8 | 11 | 20 | 32 | 1 | 5 | 8 | 9 | 10 | 11 | 12 | 20 |    |    |
| 5 | 9 | 10 | 12 | 17 | + | 8 | 11 | 20 | 32 | 1 | 5 | 8 | 9 | 10 | 11 | 12 | 17 | 20 | 32 |

L'algoritmo di ordinamento per fusione ordina un array dividendolo a metà, ordinando ricorsivamente ciascuna metà e fondendo, poi, le due metà ordinate.

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.

Tutto questo sarà anche divertente, ma non sembra che possa risolvere il problema per il computer, che si trova a dover ancora ordinare la prima e la seconda metà dell'array, perché 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 realizzi questa idea. Quando un oggetto di tipo `MergeSorter` ordina un array, esso crea due array, ciascuno avente dimensione pari alla metà dell'array originario, e li ordina ricorsivamente. Quindi, fonde insieme i due array ordinati:

```
public void sort()
{
    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
    ...
    MergeSorter firstSorter = new MergeSorter(first);
    MergeSorter secondSorter = new MergeSorter(second);
    firstSorter.sort();
    secondSorter.sort();
    merge(first, second);
}
```

Il metodo `merge` è noioso ma abbastanza semplice: lo troverete nel codice che segue.

### File ch13/mergesort/MergeSorter.java

```
/**
 * Questa classe ordina un array, usando l'algoritmo
 * di ordinamento per fusione.
 */
public class MergeSorter
{
    private int[] a;

    /**
     * Costruisce un ordinatore per fusione.
     * @param anArray l'array da ordinare
     */
    public MergeSorter(int[] anArray)
    {
        a = anArray;
    }

    /**
     * Ordina l'array gestito da questo ordinatore per fusione.
     */
    public void sort()
```

```
{  
    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  
    for (int i = 0; i < first.length; i++) { first[i] = a[i]; }  
    for (int i = 0; i < second.length; i++)  
    {  
        second[i] = a[first.length + i];  
    }  
    MergeSorter firstSorter = new MergeSorter(first);  
    MergeSorter secondSorter = new MergeSorter(second);  
    firstSorter.sort();  
    secondSorter.sort();  
    merge(first, second);  
}  
  
/**  
 * Fonde due array ordinati per generare l'array che  
 * deve essere ordinato da questo ordinatore per fusione.  
 * @param first il primo array ordinato  
 * @param second il secondo array ordinato  
 */  
private void merge(int[] first, int[] second)  
{  
    // il prossimo elemento da considerare nel primo array  
    int iFirst = 0;  
    // il prossimo elemento da considerare nel secondo array  
    int iSecond = 0;  
    // la prossima posizione libera nell'array a  
    int j = 0;  
  
    // finché né iFirst né iSecond oltrepassano la fine  
    // del relativo array, sposta in a l'elemento minore  
    while (iFirst < first.length && iSecond < second.length)  
    {  
        if (first[iFirst] < second[iSecond])  
        {  
            a[j] = first[iFirst];  
            iFirst++;  
        }  
        else  
        {  
            a[j] = second[iSecond];  
            iSecond++;  
        }  
        j++;  
    }  
  
    // notate che soltanto una delle due copiature  
    // seguenti viene eseguita  
  
    // copia in a tutti i valori rimasti nel primo array  
    while (iFirst < first.length)  
    {  
        a[j] = first[iFirst];  
    }  
}
```

```
iFirst++; j++;
}
// copia in a tutti i valori rimasti nel secondo array
while (iSecond < second.length)
{
    a[j] = second[iSecond];
    iSecond++; j++;
}
}
```

## File ch13/mergesort/MergeSortDemo.java

```
import java.util.Arrays;

/**
 * Questo programma applica l'algoritmo di ordinamento
 * per fusione a un array riempito con numeri casuali.
 */
public class MergeSortDemo
{
    public static void main(String[] args)
    {
        int[] a = ArrayUtil.randomIntArray(20, 100);
        System.out.println(Arrays.toString(a));

        MergeSorter sorter = new MergeSorter(a);
        sorter.sort();
        System.out.println(Arrays.toString(a));
    }
}
```

## 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, 76, 81, 89, 90, 98]



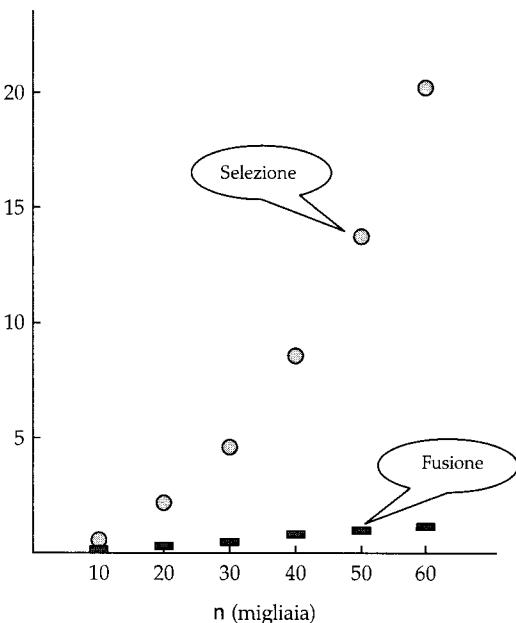
## Auto-valutazione

- Perché soltanto uno dei due cicli `while` presenti al termine del metodo `merge` fa qualcosa?
  - Eseguite manualmente l'algoritmo di ordinamento per fusione sull'array 8 7 6 5 4 3 2 1.

## 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.

tempo (secondi)

**Figura 2**

Tempo di esecuzione  
dell'ordinamento  
per fusione (in basso)  
e per selezione (in alto).

La Figura 2 mostra una tabella e un grafico che mettono a confronto i due insiemi di misurazioni delle prestazioni: come si può vedere, 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 come prima cosa il processo di fusione che si effettua dopo che la prima e la seconda metà sono state ordinate.

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* agli 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 semi-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 semi-array si possono dividere in due parti uguali.

Sfortunatamente, la formula

$$T(n) = 2T(n/2) + 5n$$

| $n$    | Fusione<br>(millisecondi) | Selezione<br>(millisecondi) |
|--------|---------------------------|-----------------------------|
| 10 000 | 40                        | 786                         |
| 20 000 | 73                        | 2148                        |
| 30 000 | 134                       | 4796                        |
| 40 000 | 170                       | 9192                        |
| 50 000 | 192                       | 13 321                      |
| 60 000 | 205                       | 19 299                      |

non ci fornisce con chiarezza la relazione fra  $n$  e  $T(n)$ . Per capire tale relazione, valutiamo  $T(n/2)$  usando la stessa formula, ottenendo:

$$T(n/2) = 2T(n/4) + 5n/2$$

Quindi

$$T(n) = 2 \times 2T(n/4) + 5n + 5n$$

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'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 richiedeva  $100^2 = 10\ 000$  volte il tempo che necessario per ordinare 10 000 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 10 000 numeri, l'ordinamento per fusione impieghi lo stesso tempo che impiega l'ordinamento per selezione.

**L'ordinamento per fusione  
è un algoritmo  $O(n \log(n))$ .  
La funzione  $n \log(n)$  cresce  
molto più lentamente di  $n^2$ .**

cioè tre quarti di secondo sulla nostra macchina di prova (in realtà, è molto più veloce). Allora impiegherebbe circa  $0.75 \times 150$  secondi, ovvero meno di 2 minuti, per ordinare un milione di numeri interi: 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

9. 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 100 000 valori?
10. Se raddoppiate la dimensione di un array, come aumenta il tempo richiesto per ordinare il nuovo array usando l'algoritmo di ordinamento per fusione?

## 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* in inglese, “divide et impera” in latino). Per ordinare la porzione `a[from] ... a[to]` dell'array `a`, si dispongono dapprima gli elementi in modo che nessuno di quelli presenti nella parte `a[from] ... a[p]` sia maggiore di elementi dell'altra parte, `a[p + 1] ... 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 lo stesso algoritmo. Ciò ordina l'intera porzione originaria, perché il maggiore elemento della prima parte è al massimo uguale al minore elemento della seconda parte.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|

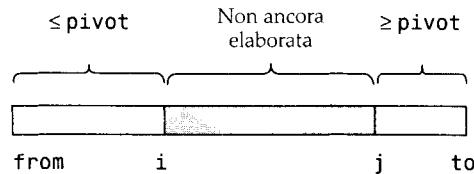
Ecco un'implementazione ricorsiva di quicksort:

```
public void sort(int from, int to)
{
    if (from >= to) return;
    int p = partition(from, to);
    sort(from, p);
    sort(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,  $a[from]$ .

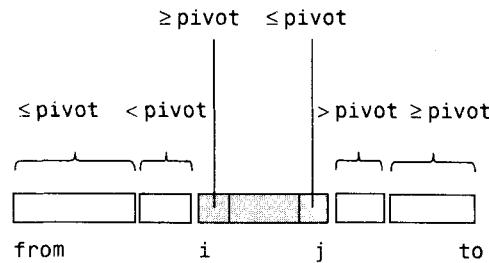
Ora create due regioni:  $a[from] \dots a[i]$ , contenente valori non maggiori del pivot, e  $a[j] \dots a[to]$ , contenente valori non minori del pivot. La regione  $a[i + 1] \dots a[j - 1]$  contiene valori che non sono ancora stati analizzati. All'inizio, le zone sinistra e destra sono vuote, cioè  $i = from - 1$  e  $j = to + 1$ .

Suddivisione  
di una porzione  
di array da ordinare



A questo punto, incrementate  $i$  finché  $a[i] < pivot$  e decrementate  $j$  finché  $a[j] > pivot$ . La figura mostra  $i$  e  $j$  quando il procedimento si arresta.

Estensione delle due parti,  
sinistra e destra



Ora scambiate i valori che si trovano nelle posizioni  $i$  e  $j$ , estendendo ancora una volta entrambe le zone, e proseguite finché  $i < j$ . Ecco il codice per il metodo *partition*:

```
private int partition(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) swap(i, j);
    }
    return j;
}

```

In media, l'algoritmo quicksort ha prestazioni  $O(n \log(n))$  e, nella maggior parte dei casi, viene eseguito più velocemente dell'ordinamento per fusione, perché è più semplice. C'è un solo aspetto sfortunato nell'algoritmo quicksort: il suo comportamento nel *caso peggiore* è  $O(n^2)$ . Inoltre, se come elemento 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 con più cura l'elemento pivot, 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 7.

## 13.6 Effettuare ricerche

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 è in ordine alfabetico. Pensate, ora, di avere un numero di telefono e di voler sapere a chi è intestato: 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* o *sequenziale*.

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, cercando il valore `v`: il metodo `search` restituisce l'indice della prima corrispondenza trovata, oppure `-1` se `v` non è presente in `a`.

**La ricerca lineare esamina tutti i valori di un array finché trova una corrispondenza con quanto cercato oppure raggiunge la fine dell'array.**

**La ricerca lineare trova un valore in un array in  $O(n)$  passi.**



## Note di cronaca 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 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, a destra, le *differenze* fra valori consecutivi:

|     |    |
|-----|----|
| 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 |
| 27  | 18 |
| 64  | 24 |
| 125 | 30 |
| 216 | 91 |

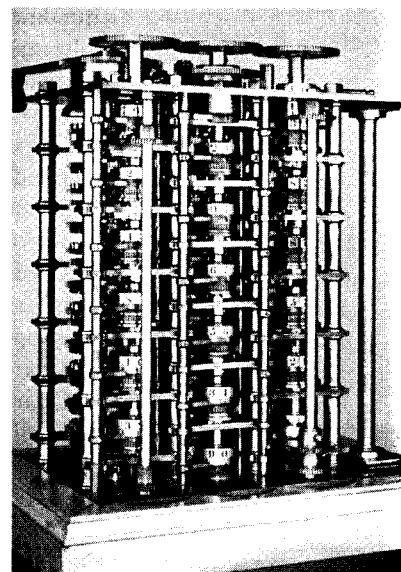
Ora le differenze sono costanti. Potete ritrovare i valori della funzione mediante una sequenza di addizioni: dovete conoscere la differenza costante e i valori che si trovano sul bordo superiore dello schema. Potete provare: scrivete su un foglio di carta i numeri evidenziati, riempiendo le posizioni 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 addizionali meccaniche già si conoscevano da tempo: erano costituite da ruote dentate, con dieci denti per ruota che rappresentavano 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 decorate.

Ada Augusta, contessa di Lovelace (1815-1852), unica figlia di

### Difference Engine di Babbage



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 della storia. Il linguaggio di programmazione Ada, un linguaggio

sviluppato per essere utilizzato nei progetti del Dipartimento della Difesa degli USA, è stato così chiamato in suo onore.

### File ch13/linsearch/LinearSearcher.java

```
/*
 * Una classe per eseguire ricerche lineari in un array.
 */
public class LinearSearcher
{
    private int[] a;

    /**
     * Costruisce un oggetto di tipo LinearSearcher.
     * @param anArray un array di numeri interi
     */
    public LinearSearcher(int[] anArray)
    {
        a = anArray;
    }

    /**
     * Cerca un valore in un array usando l'algoritmo
     * di ricerca lineare.
     * @param v il valore da cercare
     * @return l'indice in cui si trova il valore, oppure -1
     *         se il valore non è presente nell'array
     */
    public int search(int v)
    {
        for (int i = 0; i < a.length; i++)
        {
            if (a[i] == v)
                return i;
        }
        return -1;
    }
}
```

### File ch13/linsearch/LinearSearchDemo.java

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

/**
 * Questo programma utilizza l'algoritmo di ricerca lineare.
 */
public class LinearSearchDemo
{
```

```

public static void main(String[] args)
{
    int[] a = ArrayUtil.randomIntArray(20, 100);
    System.out.println(Arrays.toString(a));
    LinearSearcher searcher = new LineareSearcher(a);

    Scanner in = new Scanner(System.in);

    boolean done = false;
    while (!done)
    {
        System.out.print("Enter number to search for, -1 to quit: ");
        int n = in.nextInt();
        if (n == -1)
            done = true;
        else
        {
            int pos = searcher.search(n);
            System.out.println("Found in position " + pos);
        }
    }
}

```

### 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

```



### Auto-valutazione

11. 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?
12. Perché nel metodo `search` non si può usare un ciclo generalizzato come `for (int element : a)`?

## 13.7 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] |
| 1   | 5   | 8   | 9   | 12  | 17  | 20  | 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[3]`, è 9: è più piccolo del valore che stiamo cercando.

quindi dovremo cercare nella seconda metà dell'insieme, cioè nella porzione evidenziata in grigio

|                     |
|---------------------|
| + 5 8 9 12 17 20 32 |
| + 5 8 9 12 17 20 32 |

Ora, l'ultimo valore della prima metà di questa porzione è 17, quindi il valore che cerchiamo deve essere localizzato nella zona qui evidenziata

|                     |
|---------------------|
| + 5 8 9 12 17 20 32 |
| + 5 8 9 12 17 20 32 |

L'ultimo valore della prima metà di questa brevissima porzione è 12, che è più piccolo del valore che stiamo cercando, per cui dobbiamo esaminare la seconda metà

|                     |
|---------------------|
| + 5 8 9 12 17 20 32 |
| + 5 8 9 12 17 20 32 |

È banale constatare che non abbiamo trovato il numero cercato, perché  $15 \neq 17$ . Se volessimo inserire 15 nella sequenza, dovremmo inserirlo appena prima di  $a[5]$ .

Questo processo di ricerca si chiama *ricerca binaria* o *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 `v` non viene trovato in `a`.

### File ch13/binsearch/BinarySearcher.java

```
/*
 * Una classe per eseguire ricerche binarie in un array.
 */
public class BinarySearcher
{
    private int[] a;

    /**
     * Costruisce un oggetto di tipo BinarySearcher.
     * @param anArray un array ordinato di numeri interi
     */
    public BinarySearcher(int[] anArray)
    {
        a = anArray;
    }

    /**
     * Cerca un valore in un array ordinato,
     * utilizzando l'algoritmo di ricerca binaria.
     * @param v il valore da cercare
     * @return l'indice in cui si trova il valore, oppure -1
     *         se il valore non è presente nell'array
    
```

La ricerca binaria cerca un valore in un array ordinato determinando se il valore si trova nella prima o nella seconda metà dell'array, ripetendo poi la ricerca in una delle due metà.

```

*/
public int search(int v)
{
    int low = 0;
    int high = a.length - 1;
    while (low <= high)
    {
        int mid = (low + high) / 2;
        int diff = a[mid] - v;

        if (diff == 0) // a[mid] == v
            return mid;
        else if (diff < 0) // a[mid] < v
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

```

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 semi-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:

$$T(n) = T(n/4) + 2$$

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,  $n = 2^m$ , dove  $m$  è  $\log_2(n)$ . Otteniamo, quindi

$$T(n) = 1 + \log_2(n)$$

Di conseguenza, la ricerca binaria è un algoritmo  $O(\log(n))$ .

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: queste

coincide con la nostra formula, dal momento che  $\log_2(100) \approx 6.64386$  e, in effetti, la più piccola potenza intera di 2 maggiore di 100 è  $2^7 = 128$ .

Dal momento che la ricerca binaria è tanto più veloce della ricerca lineare, vale la pena effettuare prima l'ordinamento di un array non ordinato, per poi ricorrere alla ricerca binaria? Dipende. Se nell'array effettuate 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.

La classe `Arrays` contiene un metodo statico, `binarySearch`, che realizza l'algoritmo di ricerca binaria, a dire il vero con un utile miglioramento: se un valore non viene trovato nell'array, il valore restituito non è  $-1$ , ma  $-k-1$ , dove  $k$  è la posizione prima della quale andrebbe inserito l'elemento per mantenere ordinato l'array. Ad esempio:

```
int[] a = { 1, 4, 9 };
int v = 7;
int pos = Arrays.binarySearch(a, v);
// restituisce -3; v andrebbe inserito prima della posizione 2
```

## Auto-valutazione

13. Immaginate di dover cercare un valore in un array ordinato di un milione elementi. Usando l'algoritmo di ricerca binaria, quanti elementi pensate di dover esaminare, mediamente, per trovare il valore che cercate?
14. Perché è utile che il metodo `Arrays.binarySearch` segnali la posizione in cui andrebbe inserito un elemento che non è stato trovato?
15. Perché, per segnalare che un valore non è presente e che andrebbe inserito prima della posizione  $k$ , `Arrays.binarySearch` restituisce  $-k-1$  e non  $-k$ ?

## 13.8 Ordinare dati veri

**La classe `Arrays` contiene un metodo di ordinamento.**

Quando scrivete programmi in Java non avete bisogno di realizzare algoritmi di ordinamento: la classe `Arrays` contiene un metodo statico, `sort`, che è 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: consultate Argomenti avanzati 13.3 per avere maggior informazioni in merito.

Naturalmente, nella programmazione reale è raro che si debbano fare ricerche in insiemi di numeri interi, ma è facile modificare queste tecniche per fare ricerche in collezioni di dati reali.

La classe `Arrays` contiene anche un metodo statico `sort` che è in grado di ordinare array di oggetti, ma la classe `Arrays` non sa come confrontare oggetti di tipo arbitrario. Immaginate, ad esempio, di voler ordinare array di oggetti di tipo `Coin`: potreste ordinarli in base al loro nome oppure in base al loro valore. Il metodo `Arrays.sort` non può

**Il metodo `sort` della classe `Arrays` ordina oggetti di classi che realizzano l'interfaccia `Comparable`.**

prendere questa decisione per voi, per cui richiede che gli oggetti appartengano a classi che realizzano l'interfaccia `Comparable`, che ha un unico metodo:

```
public interface Comparable
{
    int compareTo(Object otherObject);
}
```

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.

Molte classi della libreria standard di Java, come `String` e `Date`, realizzano l'interfaccia `Comparable`.

Potete realizzare l'interfaccia `Comparable` anche nelle vostre classi. Per esempio, per ordinare una raccolta di monete, la classe `Coin` dovrebbe realizzare tale interfaccia, dichiarando un metodo `compareTo`:

```
public class Coin implements Comparable
{
    . .
    public int compareTo(Object otherObject)
    {
        Coin other = (Coin) otherObject;
        if (value < other.value) return -1;
        if (value == other.value) return 0;
        return 1;
    }
    . .
}
```

Quando realizzate il metodo `compareTo` dell'interfaccia `Comparable`, dovete essere certi che il metodo definisca una *relazione d'ordine totale*, con le tre seguenti proprietà:

- *Antisimmetrica*: se `a.compareTo(b) ≤ 0`, allora `b.compareTo(a) ≥ 0`
- *Riflessiva*: `a.compareTo(a) = 0`
- *Transitiva*: se `a.compareTo(b) ≤ 0` e `b.compareTo(c) ≤ 0`, allora `a.compareTo(c) ≤ 0`

Una volta che la vostra classe `Coin` realizza l'interfaccia `Comparable`, potete semplicemente usare un array di monete come argomento del metodo `Arrays.sort`:

```
Coin[] coins = new Coin[n];
// aggiungi monete
...
Arrays.sort(coins);
```

La classe `Collections` contiene un metodo `sort` che è in grado di ordinare vettori.

Se le monete sono memorizzate in un oggetto di tipo `ArrayList`, utilizzate invece il metodo `Collections.sort`, che usa l'algoritmo di ordinamento per fusione:

```
ArrayList<Coin> coins = new ArrayList<Coin>();
// aggiungi monete
...
Collections.sort(coins);
```

In pratica, non dovreste usare metodi di ordinamento e ricerca scritti da voi, bensì quelli presenti nelle classi `Arrays` e `Collections`. Gli algoritmi della libreria sono stati ben collaudati e ottimizzati, per cui 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

16. Perché il metodo `Arrays.sort` non può ordinare un array di oggetti di tipo `Rectangle`?
17. Cosa bisogna fare per ordinare in ordine di saldo crescente un array di oggetti di tipo `BankAccount`?



## Errori comuni 13.1

### Il metodo `compareTo` può restituire qualsiasi valore intero, non solo -1, 0 o 1

L'invocazione `a.compareTo(b)`, per 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 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`.



## Argomenti avanzati 13.4

### L'interfaccia Comparable parametrica

A partire dalla versione 5 di Java, l'interfaccia Comparable è un tipo parametrico, simile ad ArrayList:

```
public interface Comparable<T>
{
    int compareTo(T other);
}
```

Il tipo parametrico, T, specifica il tipo degli oggetti che viene accettato da una determinata classe per fare confronti e, solitamente, si tratta della classe stessa. Ad esempio, la classe Coin realizzerà probabilmente l'interfaccia Comparable<Coin>, in questo modo:

```
public class Coin implements Comparable<Coin>
{
    .
    .
    public int compareTo(Coin other)
    {
        if (value < other.value) return -1;
        if (value == other.value) return 0;
        return 1;
    }
    .
}
```

Usare un tipo parametrico come parametro del metodo compareTo ha un vantaggio davvero significativo: non c'è bisogno di usare un cast per convertire un parametro di tipo Object nel tipo desiderato.



## Argomenti avanzati 13.5

### L'interfaccia Comparator

A volte si vuole ordinare un array o un vettore di oggetti che non appartengono a una classe che realizza 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 monete per nome invece che per valore.

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 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 realizzzi l'interfaccia Comparator, che ha un unico metodo, compare, che confronta due oggetti.

A partire dalla versione 5 di Java, l'interfaccia Comparator è un tipo parametrico, il cui parametro specifica il tipo dei parametri ricevuti dal metodo compare. Ad esempio, l'interfaccia Comparator<Coin> è questa:

```
public interface Comparator<Coin>
{
    int compare(Coin a, Coin b);
}
```

Se `comp` è un esemplare di una classe che realizza `Comparator<Coin>`, 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 monete:

```
public class CoinComparator implements Comparator<Coin>
{
    public int compare(Coin a, Coin b)
    {
        if (a.getValue() < b.getValue()) return -1;
        if (a.getValue() == b.getValue()) return 0;
        return 1;
    }
}
```

Per ordinare un array di monete, `coins`, in base al loro valore, invocate:

```
Arrays.sort(coins, new CoinComparator());
```

## 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 minore della regione terminale non ancora ordinata, spostandolo all'inizio della regione 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.

### Utilizzo della notazione O-grande per descrivere il tempo di esecuzione di un algoritmo

- Gli informatici teorici usano la notazione O-grande: se  $f(n) = O(g(n))$ , la funzione  $f$  non aumenta più rapidamente della funzione  $g$ .
- 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 metà e fondendo, poi, le due metà ordinate.

### Confronto dei 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  $n^2$ .

### Descrizione dell'algoritmo di ricerca lineare e il suo tempo di esecuzione

- La ricerca lineare esamina tutti i valori di un array finché trova una corrispondenza con quanto cercato oppure raggiunge la fine dell'array.
- La ricerca lineare trova un valore in un array in  $O(n)$  passi.

### Descrizione dell'algoritmo di ricerca binaria e il suo tempo di esecuzione

- La ricerca binaria cerca un valore in un array ordinato determinando se il valore si trova 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  $O(\log(n))$  passi.

### Utilizzo dei metodi della libreria di Java per ordinare dati

- La classe `Arrays` contiene un metodo di ordinamento.
- Il metodo `sort` della classe `Arrays` ordina oggetti di classi che realizzano l'interfaccia `Comparable`.
- La classe `Collections` contiene un metodo `sort` che è in grado di ordinare vettori.

## Classi, oggetti e metodi presentati nel capitolo

|                                            |                                            |
|--------------------------------------------|--------------------------------------------|
| <code>java.lang.Comparable&lt;T&gt;</code> | <code>toString</code>                      |
| <code>compareTo</code>                     | <code>java.util.Collections</code>         |
| <code>java.lang.System</code>              | <code>binarySearch</code>                  |
| <code>currentTimeMillis</code>             | <code>sort</code>                          |
| <code>java.util.Arrays</code>              | <code>java.util.Comparator&lt;T&gt;</code> |
| <code>binarySearch</code>                  | <code>compare</code>                       |
| <code>sort</code>                          |                                            |

## Esercizi di ripasso

- ★ **Esercizio R13.1.** Qual è la differenza fra cercare e ordinare?
- ★★ **Esercizio R13.2.** Attenzione al rischio di errori per scarto di 1. Scrivendo l'algoritmo di ordinamento per selezione visto nel Paragrafo 13.1, occorre, come al solito, scegliere `<` oppure `<=`, `a.length` oppure `a.length - 1`, `from` oppure `from + 1`: un fertile terreno di coltura per la proliferazione degli errori per scarto di 1. Eseguite passo per passo il codice dell'algoritmo applicato ad array di lunghezza 0, 1, 2 e 3 e controllate accuratamente che tutti i valori degli indici siano corretti.
- ★★ **Esercizio R13.3.** Qual è l'andamento di crescita, in termini di O-grande, di queste funzioni?
  - $n^2 + 2n + 1$
  - $n^{10} + 9n^9 + 20n^8 + 145n^7$
  - $(n + 1)^4$
  - $(n^2 + n)^2$
  - $n + 0.001n^3$
  - $n^3 - 1000n^2 + 10^9$
  - $n + \log(n)$

- h.  $n^2 + n \log(n)$
- i.  $2^n + n^2$
- j.  $(n^3 + 2n)/(n^2 + 0.75)$

- \* **Esercizio 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

$$\begin{aligned} f(2000)/f(1000) \\ f(4000)/f(1000) \\ f(10000)/f(1000) \end{aligned}$$

- \* **Esercizio R13.5.** Supponiamo che un algoritmo  $O(n)$  impieghi 5 secondi per gestire un insieme di 1000 dati. Quanto tempo impiegherà per gestire un insieme di 2000 dati? E uno di 10 000?
- \*\* **Esercizio R13.6.** Supponiamo che un algoritmo impieghi 5 secondi per gestire 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       |          |               |          |
| 10 000 |        |          |          |               |          |

Per esempio, dal momento che  $3000^2/1000^2 = 9$ , se l'algoritmo fosse  $O(n^2)$ , per gestire un insieme di 3000 dati impiegherebbe un tempo 9 volte superiore a quello necessario per gestire un insieme di 1000 dati, cioè 45 secondi.

- \*\* **Esercizio R13.7.** Ordinate le seguenti espressioni O-grande in ordine crescente.

$$\begin{aligned} O(n) \\ O(n^3) \\ O(n^n) \\ O(\log(n)) \\ O(n^2 \log(n)) \\ O(\sqrt{n}) \\ O(n \log(n)) \\ O(2^n) \\ O(n\sqrt{n}) \\ O(n^{\log(n)}) \end{aligned}$$

- \* **Esercizio 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?
- \* **Esercizio R13.9.** Qual è l'andamento del tempo di esecuzione del seguente metodo?

```
public static int count(int[] a, int c)
{
    int count = 0;
    for (int i = 0; i < a.length; i++)
    {
        if (a[i] == c) count++;
    }
    return count;
}
```

- \*\* **Esercizio R13.10.** 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 dovrebbe 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?

- \*\* **Esercizio R13.11.** Considerate il seguente algoritmo per eliminare 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, eliminatelo. Questo algoritmo è più veloce di quello dell'esercizio precedente?
- \*\*\* **Esercizio R13.12.** 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.
- \*\*\* **Esercizio R13.13.** Perché, quando l'array è già ordinato, l'algoritmo di ordinamento per inserimento è significativamente più veloce dell'ordinamento per selezione?
- \*\*\* **Esercizio R13.14.** Considerate la seguente modifica, che migliora le prestazioni dell'algoritmo di ordinamento per inserimento visto in Argomenti avanzati 13.1: 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?

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

# 14

## Introduzione alle strutture di dati

### Obiettivi del capitolo

- Imparare a utilizzare le liste concatenate della libreria standard
- Saper usare gli iteratori per scandire liste concatenate
- Capire la realizzazione di liste concatenate
- Comprendere la differenza tra tipi di dati astratti e concreti
- Conoscere l'efficienza delle operazioni fondamentali per liste e array
- Acquisire familiarità con pile e code

Per contenere raccolte di oggetti, fino a questo punto abbiamo usato array come una soluzione “adatta a tutte le situazioni”, ma gli scienziati dell’informazione hanno sviluppato molte diverse strutture per i dati, caratterizzate da vari compromessi in relazione alle proprie prestazioni. In questo capitolo conoscerete la *lista concatenata* (“linked list”) o *catena*, una struttura che consente di aggiungere e rimuovere elementi in modo efficiente, senza spostare gli elementi già presenti. Comprenderete anche la differenza tra i tipi di dati concreti e astratti: un tipo astratto annuncia quali operazioni fondamentali dovrebbero essere realizzate in modo efficiente, ma non ne specifica la realizzazione. I tipi di dati pila e coda, presentati alla fine di questo capitolo, sono esempi di tipi di dati astratti.

## 14.1 Utilizzare liste concatenate

Come contenitore per una sequenza di oggetti, una *lista concatenata* (“linked list”) o *catena* è una struttura che consente di aggiungere e di rimuovere in modo efficiente elementi in qualsiasi posizione, anche intermedia, della sequenza.

Per capire per quale motivo ci sia bisogno di una struttura di questo tipo, immaginate un programma che gestisca una sequenza di oggetti che rappresentano dipendenti ordinati in base al cognome del dipendente. Quando viene assunto un nuovo dipendente, bisogna inserire un oggetto nella sequenza: a meno che non succeda che la società assuma casualmente dipendenti rispettando l’ordine alfabetico, il nuovo oggetto deve essere probabilmente inserito nella sequenza in qualche posizione intermedia, per cui tutti gli altri oggetti che seguono il nuovo dipendente devono essere spostati verso la fine della sequenza.

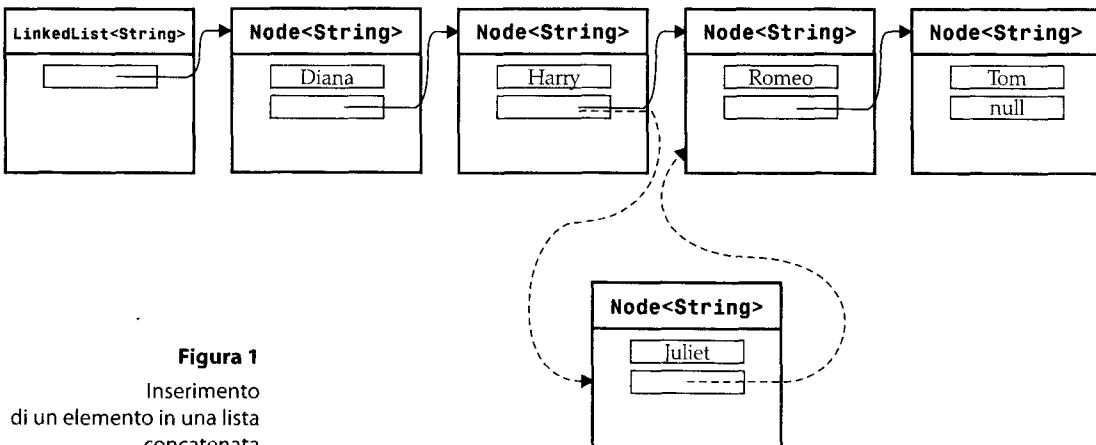
Al contrario, se un dipendente lascia la società, bisogna eliminare l’oggetto corrispondente e il buco nella sequenza va colmato spostando tutti gli oggetti che vengono dopo. Spostare un elevato numero di oggetti può comportare un notevole dispendio di tempo di elaborazione: ci piacerebbe trovare un metodo che riduca al minimo questo costo.

Invece di memorizzare i valori in un array, 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 1). Quando viene inserito un nuovo nodo in una lista concatenata, così come quando si elimina un nodo, devono essere aggiornati soltanto i riferimenti nei nodi vicini. Dove sta l’insidia? Le liste concatenate consentono inserimenti ed eliminazioni veloci, ma l’accesso agli elementi può essere lento.

Se, ad esempio, volete localizzare il quinto elemento, dovete prima visitare i primi quattro; se avete la necessità di accedere agli elementi in ordine casuale, questo è un problema. Il termine “accesso casuale” (*random access*) viene usato in informatica per descrivere una situazione 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. Ad esempio, una ricerca binaria richiede l’accesso casuale, mentre una ricerca lineare necessita di accesso sequenziale.

**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 lista concatenata è un’operazione efficiente, anche in posizioni intermedie.**



**Figura 1**  
Inserimento  
di un elemento in una lista  
concatenata

Visitare in sequenza gli elementi di una lista concatenata è efficiente, ma accedervi in ordine casuale non lo è.

Ovviamente, se visitate abitualmente tutti gli elementi 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 principalmente interessati all'efficienza degli inserimenti e delle eliminazioni e non avete bisogno di accedere agli elementi in modo casuale.

La libreria di Java mette a disposizione una classe, che imparerete a conoscere in questo paragrafo, che realizza una lista concatenata. Nel paragrafo successivo "sbircerete sotto il cofano" e vedrete come sono realizzati i suoi metodi principali.

La classe `LinkedList` del 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: `LinkedList<String>` oppure `LinkedList<Product>`.

I metodi elencati nella Tabella 1 forniscono accesso diretto sia al primo sia all'ultimo elemento della lista.

**Tabella 1**

Alcuni metodi di `LinkedList`

|                                                                             |                                                                                                                                                                                                                                                              |
|-----------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>LinkedList&lt;String&gt; lst = new LinkedList&lt;String&gt;();</code> | Una lista vuota.                                                                                                                                                                                                                                             |
| <code>lst.addLast("Harry")</code>                                           | Aggiunge un elemento alla fine della lista, esattamente come <code>add</code> .                                                                                                                                                                              |
| <code>lst.addFirst("Sally")</code>                                          | Aggiunge un elemento all'inizio della lista, dopodiché il contenuto di <code>lst</code> è <code>[Sally, Harry]</code> .                                                                                                                                      |
| <code>lst.getFirst()</code>                                                 | Restituisce l'elemento memorizzato all'inizio della lista, in questo caso <code>"Sally"</code> .                                                                                                                                                             |
| <code>lst.getLast()</code>                                                  | Restituisce l'elemento memorizzato alla fine della lista, in questo caso <code>"Harry"</code> .                                                                                                                                                              |
| <code>String removed = lst.removeFirst();</code>                            | Elimina il primo elemento della lista e lo restituisce, dopodiché il contenuto di <code>lst</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 = lst.listIterator()</code>           | Restituisce un iteratore per visitare tutti gli elementi della lista (per il suo funzionamento si veda la Tabella 2).                                                                                                                                        |

Come si fa a inserire e a togliere elementi in una posizione intermedia di una lista? La lista non vi fornisce i riferimenti ai nodi: se fosse possibile accedervi direttamente e vi capitasse di ingarbugliarli, spezzereste la concatenazione della lista. Come vedrete nel prossimo paragrafo, dove realizzerete voi stessi alcune operazioni della lista concatenata, mantenere intatti tutti i collegamenti tra i nodi non è cosa da poco.

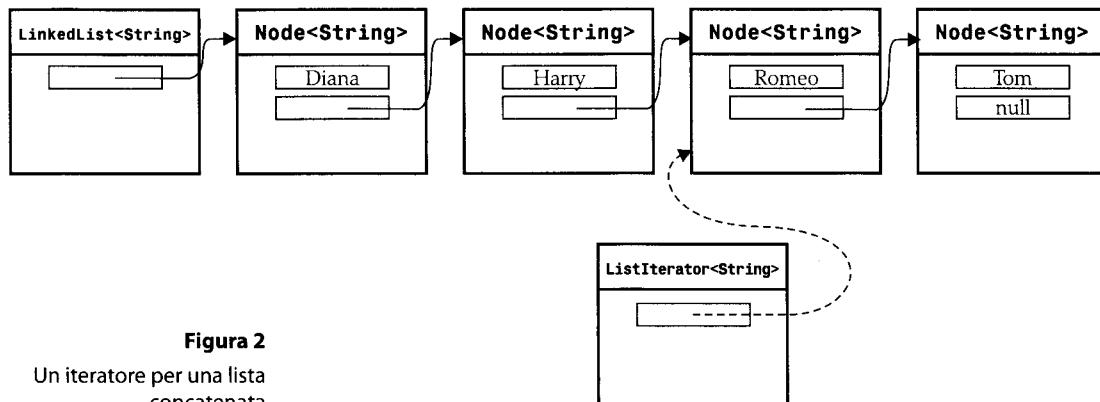
Per risolvere questo problema, la libreria di Java usa il tipo `ListIterator`, un *iteratore* (o *cursor*), che rappresenta il concetto di posizione in un qualsiasi punto entro la lista concatenata (vedi Figura 2).

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 (vedi Figura 3). In termini astratti, pensate a ciascun elemento come se fosse una lettera in un elaboratore di testi, mentre l'iteratore è il cursore intermittente che si trova fra le lettere.

Per creare un iteratore che operi su una lista utilizzate il metodo `listIterator` della classe `LinkedList`:

```
LinkedList<String> employeeNames = . . .;
ListIterator<String> iterator = employeeNames.listIterator();
```

Per accedere agli elementi presenti in una lista concatenata si usa un iteratore (o cursore).

**Figura 2**

Un iteratore per una lista concatenata

Noteate che anche la classe dell'iteratore è una classe generica: un oggetto di tipo `ListIterator<String>` scandisce le posizioni all'interno di una lista concatenata di stringhe, mentre un oggetto di tipo `ListIterator<Product>` visita gli elementi di una lista concatenata di tipo `LinkedList<Product>`.

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.

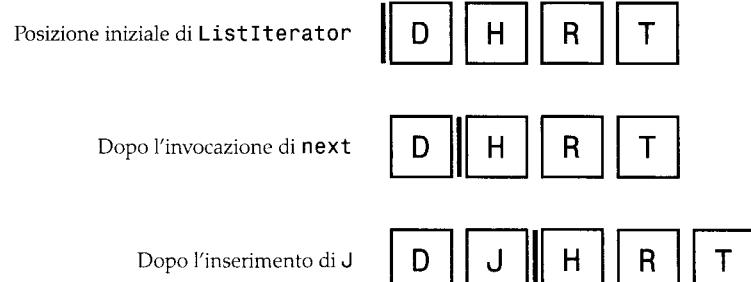
```
if (iterator.hasNext())
    iterator.next();
```

Il metodo `next` restituisce l'elemento sopra cui transita l'iteratore durante il suo avanzamento: quando usate un iteratore di tipo `ListIterator<String>`, il suo 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.

Potete visitare tutti gli elementi di una lista concatenata di stringhe usando questo ciclo:

**Figura 3**

Una visione astratta dell'iteratore operante su una lista



```

while (iterator.hasNext())
{
    String name = iterator.next();
    Fa qualcosa con name
}

```

Più sinteticamente, se il vostro ciclo deve semplicemente visitare tutti gli elementi di una lista concatenata, potete usare un ciclo `for` esteso:

```

for (String name : employeeNames)
{
    Fa qualcosa con 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 (vedere Argomenti avanzati 14.1).

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 usare i metodi `previous` e `hasPrevious` dell'interfaccia `ListIterator` per spostare l'iteratore all'indietro.

Il metodo `add` aggiunge un oggetto 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 word processor: 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 3). Gran parte delle persone non ci fa caso: fate una prova e osservate attentamente in che modo il vostro word processor 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();
}

```

Dovete prestare molta attenzione quando invocate `remove`, perché può essere invocato una sola volta dopo aver invocato `next` o `previous`, quindi questo è un errore:

```

iterator.next();
iterator.next();
iterator.remove();
iterator.remove() // ERRORE: non si può invocare remove due volte

```

Inoltre, non potete invocarlo immediatamente dopo add:

```
iterator.add("Fred");
iterator.remove() // ERRORE: invocare remove solo dopo next o previous
```

Se invocato in modo improprio, il metodo lancia una eccezione di tipo `IllegalStateException`.

La Tabella 2 riassume i metodi dell'interfaccia `ListIterator`.

| <b>Tabella 2</b><br>Alcuni metodi<br>di <code>ListIterator</code>                   |                                                                                                                                                                                                                                                                        |
|-------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>String s = iter.next();</code>                                                | Ipotizziamo che, prima di questa invocazione di <code>next</code> , <code>iter</code> punti all'inizio della lista, il cui contenuto sia <code>[Sally]</code> . Dopo l'esecuzione, <code>s</code> vale <code>"Sally"</code> e l'iteratore punta alla fine della lista. |
| <code>iter.hasNext()</code>                                                         | Restituisce <code>false</code> perché l'iteratore si trova ora alla fine della lista.                                                                                                                                                                                  |
| <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.                                                                                                                                               |
| <code>iter.add("Diana")</code>                                                      | Aggiunge un elemento prima della posizione dell'iteratore, dopodiché il contenuto della lista è <code>[Diana, Sally]</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> . Il contenuto della lista è nuovamente <code>[Sally]</code> .                                                                                         |

Ecco un programma dimostrativo che inserisce stringhe in una lista e, quindi, la percorre iterativamente, aggiungendo ed eliminando elementi. Alla fine, viene visualizzata l'intera lista. I commenti segnalano la posizione dell'iteratore.

### File ch14/uselist/ListTester.java

```
import java.util.LinkedList;
import java.util.ListIterator;

/**
 * Programma che illustra il funzionamento della classe LinkedList.
 */
public class ListTester
{
    public static void main(String[] args)
    {
        LinkedList<String> staff = new LinkedList<String>();
        staff.addLast("Diana");
        staff.addLast("Harry");
        staff.addLast("Romeo");
        staff.addLast("Tom");

        // il segno | nei commenti indica la posizione dell'iteratore

        ListIterator<String> iterator = staff.listIterator(); // |DHRT
        iterator.next(); // D|HRT
        iterator.next(); // DH|RT

        // aggiunge altri elementi dopo il secondo
```

```

        iterator.add("Juliet"); // DHJ|RT
        iterator.add("Nina"); // DHJN|RT

        iterator.next(); // DHJNR|T

        // toglie l'ultimo elemento attraversato

        iterator.remove(); // DHJN|T

        // visualizza tutti gli elementi

        for (String name : staff)
            System.out.print(name + " ");
        System.out.println();
        System.out.println("Expected: Diana Harry Juliet Nina Tom");
    }
}

```

## Visualizza

```

Diana Harry Juliet Nina Tom
Expected: Diana Harry Juliet Nina Tom

```

## Auto-valutazione

1. Le liste concatenate occupano più spazio in memoria degli array di ugual dimensione?
2. Perché non abbiamo bisogno di iteratori per gli array?

## Argomenti avanzati 14.1

### L'interfaccia Iterable e il ciclo for esteso

Potete usare il ciclo **for** esteso

```
for (Tipo variabile : contenitore)
```

con qualsiasi classe di tipo contenitore della libreria standard di Java, come le classi `ArrayList` e `LinkedList` e le classi di libreria che verranno illustrate nel Capitolo 15. In realtà, il ciclo **for** esteso può essere utilizzato con qualsiasi classe che realizzzi l'interfaccia `Iterable`:

```

public interface Iterable<E>
{
    Iterator<E> iterator();
}

```

L'interfaccia è parametrica e il tipo del parametro, `E`, indica il tipo degli elementi che fanno parte del contenitore. Il suo unico metodo, `iterator`, restituisce un oggetto che realizza l'interfaccia `Iterator<E>`, dotata dei metodi

```

boolean hasNext();
E next();

```

L'interfaccia `ListIterator` che avete visto nel paragrafo precedente è un'interfaccia derivata da `Iterator`, con alcuni metodi in più, come `add` e `previous`.

Il compilatore traduce un ciclo `for` esteso in un ciclo equivalente che usa un iteratore. Il ciclo

```
for (Tipo variabile : contenitore)
    corpoDelCiclo
```

è equivalente a

```
Iterator<Tipo> iter = contenitore.iterator();
while (iter.hasNext())
{
    Tipo variabile = iter.next();
    corpoDelCiclo
}
```

Le classi `ArrayList` e `LinkedList` realizzano l'interfaccia `Iterable`; se anche le vostre classi la realizzano, le potrete usare allo stesso modo in un ciclo `for` esteso (Esercizio P14.19).

## 14.2 Realizzare liste concatenate

Nel paragrafo precedente avete visto come si utilizza la classe “lista concatenata” fornita dalla libreria Java. In questo paragrafo esamineremo la realizzazione di una versione semplificata di questa classe: in questo modo potrete vedere come le operazioni applicate alla lista ne manipolano i collegamenti mentre la lista viene modificata.

Per non complicare il codice, che vuole essere solo un esempio, non realizzeremo tutti i metodi della classe che rappresenta la lista concatenata: realizzeremo soltanto una lista semplicemente concatenata e la classe fornirà l'accesso diretto soltanto al primo elemento della lista, non all'ultimo. Inoltre, la nostra classe non sarà parametrica: memorizzeremo, semplicemente, valori di tipo `Object` e inseriremo i cast opportuni quando li recupereremo dal contenitore. Il risultato sarà una classe “lista” pienamente funzionale, che mostra come vengono aggiornati i collegamenti nelle operazioni `add` e `remove` e come l'iteratore scandisce la lista.

Un oggetto di tipo `Node` (un nodo della lista) memorizza un oggetto e un riferimento al nodo successivo. Siccome sia i metodi della lista concatenata sia quelli della classe iteratore accedono di frequente alle variabili di esemplare di `Node`, non definiamo come private tali variabili, ma rendiamo `Node` una classe interna della classe `LinkedList`. Dato che nessun metodo della lista restituisce un oggetto di tipo `Node`, non ci sono rischi a lasciare pubbliche le variabili di esemplare di quest'ultima classe.

```
public class LinkedList
{
    .
    .
    class Node
    {
        public Object data;
        public Node next;
    }
}
```

Un esemplare di lista concatenata memorizza un riferimento al primo nodo, mentre ciascun nodo memorizza un riferimento al nodo successivo.

La nostra classe `LinkedList` contiene un riferimento, `first`, al primo nodo (oppure `null`, se la lista è completamente vuota).

```
public class LinkedList
{
    private Node first;
    . .
    public LinkedList()
    {
        first = null;
    }

    public Object getFirst()
    {
        if (first == null) throw new NoSuchElementException();
        return first.data;
    }
}
```

Passiamo ora al metodo `addFirst` (osservate la Figura 4). Quando viene aggiunto alla lista un nuovo nodo, questo diventa la testa (*head*) della lista e il nodo che in precedenza era la testa della lista diventa il suo successivo:

```
public class LinkedList
{
    . .

    public void addFirst(Object element)
    {
        Node newNode = new Node(); // 1 in figura
        newNode.data = element;
        newNode.next = first; // 2 in figura
        first = newNode; // 3 in figura
    }
    . .
}
```

L'eliminazione del primo elemento della lista funziona così: il dato presente nel primo nodo viene memorizzato e successivamente restituito come risultato del metodo; il successore del primo nodo diventa il primo nodo della lista accorciata (osservate la Figura 5); a questo punto non vi sono ulteriori riferimenti al vecchio nodo e il *garbage collector* lo riciclerà al momento opportuno.

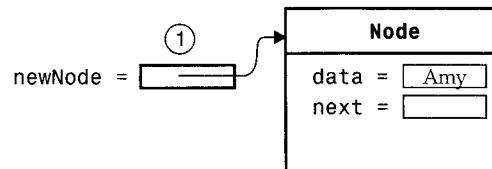
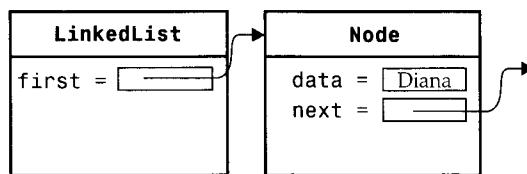
```
public class LinkedList
{
    . .

    public Object removeFirst()
    {
        if (first == null) throw new NoSuchElementException();
        Object element = first.data;
        first = first.next; // 1 in figura
        return element;
    }
    . .
}
```

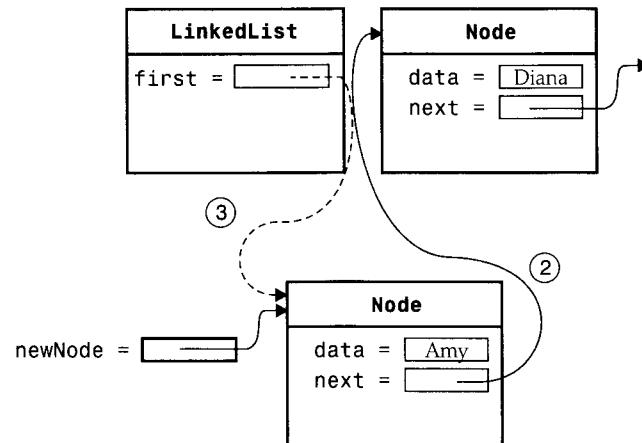
**Figura 4**

Inserimento di un nuovo nodo all'inizio di una lista concatenata.

Prima dell'inserimento

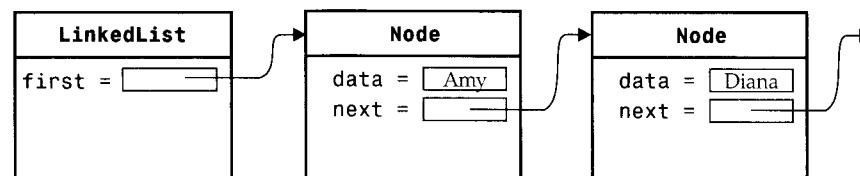


Dopo l'inserimento

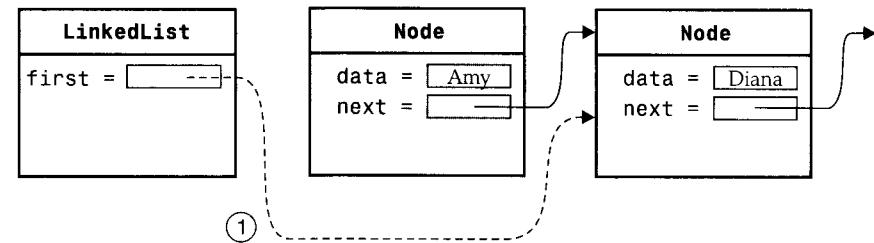
**Figura 5**

Rimozione del primo nodo di una lista concatenata.

Prima della rimozione



Dopo la rimozione



Ora dobbiamo progettare la classe iteratore. Nella libreria standard di Java, `ListIterator` è un'interfaccia che definisce nove metodi, di cui ne ometteremo quattro (i metodi che spostano l'iteratore all'indietro e i metodi che forniscono un indice numerico per rappresentare la posizione dell'iteratore).

La nostra classe `LinkedList` dichiara una classe interna, `LinkedListIterator`, che realizza l'interfaccia `ListIterator` così semplificata. Poiché `LinkedListIterator` è una classe interna, ha accesso alle caratteristiche private della classe `LinkedList`, tra cui il campo `first` e la classe `Node`.

I programmi che utilizzano la classe `LinkedList` non hanno bisogno di conoscere il nome della classe iteratore: è sufficiente sapere che è una classe che realizza l'interfaccia `ListIterator`.

```
public class LinkedList
{
    ...
    public ListIterator listIterator()
    {
        return new LinkedListIterator();
    }

    class LinkedListIterator implements ListIterator
    {
        private Node position;
        private Node previous;
        ...
        public LinkedListIterator()
        {
            position = null;
            previous = null;
        }
    }
    ...
}
```

**Un oggetto iteratore memorizza  
un riferimento all'ultimo nodo  
che ha visitato nella lista.**

Ciascun oggetto iteratore ha un riferimento, `position`, all'ultimo nodo visitato. Memorizziamo anche un riferimento al penultimo nodo, `previous`: ne avremo bisogno per sistemare opportunamente i collegamenti nel metodo `remove`.

Il metodo `next` è semplice: il riferimento `position` viene fatto progredire fino a `position.next` e la vecchia posizione viene memorizzata in `previous`. C'è, però, un caso speciale: se l'iteratore è nella posizione che precede il primo nodo della lista, il vecchio valore di `position` è `null` e deve diventare uguale a `first`.

```
class LinkedListIterator implements ListIterator
{
    ...
    public Object next()
    {
        if (!hasNext())
            throw new NoSuchElementException();
        previous = position; // memorizza per remove
        if (position == null)
```

```

        position = first;
    else
        position = position.next;

    return position.data;
}
...
}

```

È previsto che il metodo `next` venga invocato soltanto quando l'iteratore non è ancora arrivato alla fine della lista, quindi dichiariamo il metodo `hasNext` in modo coerente. L'iteratore è alla fine della lista se la questa è vuota (vale a dire, se `first == null`) oppure se non vi è alcun elemento dopo la posizione corrente (`position.next == null`).

```

class LinkedListIterator implements ListIterator
{
    ...
    public boolean hasNext()
    {
        if (position == null)
            return first != null;
        else
            return position.next != null;
    }
    ...
}

```

Il metodo `set` modifica il dato memorizzato nell'elemento visitato in precedenza: la sua realizzazione è immediata, perché le nostre liste concatenate possono essere attraversate in una sola direzione. La realizzazione di lista concatenata presente nella libreria standard deve, invece, tenere traccia della direzione dell'ultimo movimento dell'iteratore: per tale ragione, la libreria standard impedisce l'uso del metodo `set` subito dopo l'invocazione del metodo `add` o `remove`. Noi non imporremo tale restrizione, che nel nostro caso non è necessaria.

```

public void set(Object element)
{
    if (position == null)
        throw new NoSuchElementException();
    position.data = element;
}

```

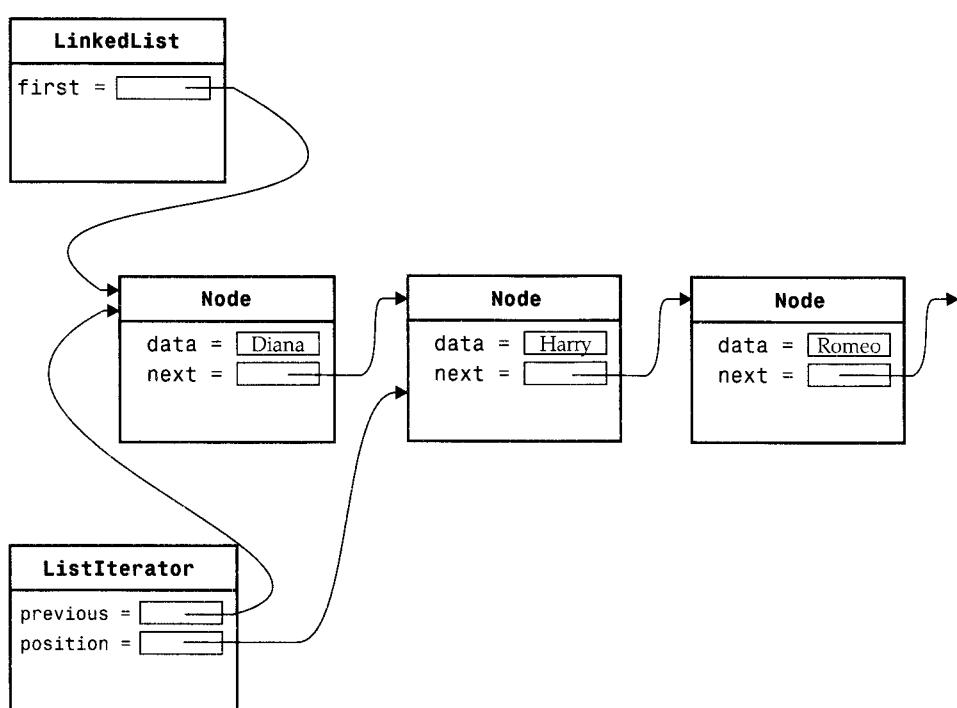
La realizzazione di operazioni che modificano una lista concatenata è una sfida: bisogna verificare con cura che vengano aggiornati in modo corretto tutti i riferimenti ai nodi.

Eliminare l'ultimo nodo visitato è un'operazione più complessa. Se l'elemento che deve essere rimosso è il primo, invochiamo semplicemente `removeFirst`. In caso contrario, l'elemento da rimuovere si trova all'interno della lista, per cui si dovrà aggiornare il riferimento `next` del nodo che lo precede in modo da saltare l'elemento rimosso (osservate la Figura 6). Se il riferimento `previous` è uguale a `position`, allora questa invocazione di `remove` non segue immediatamente un'invocazione di `next` e lanciamo un'eccezione di tipo `IllegalStateException`.

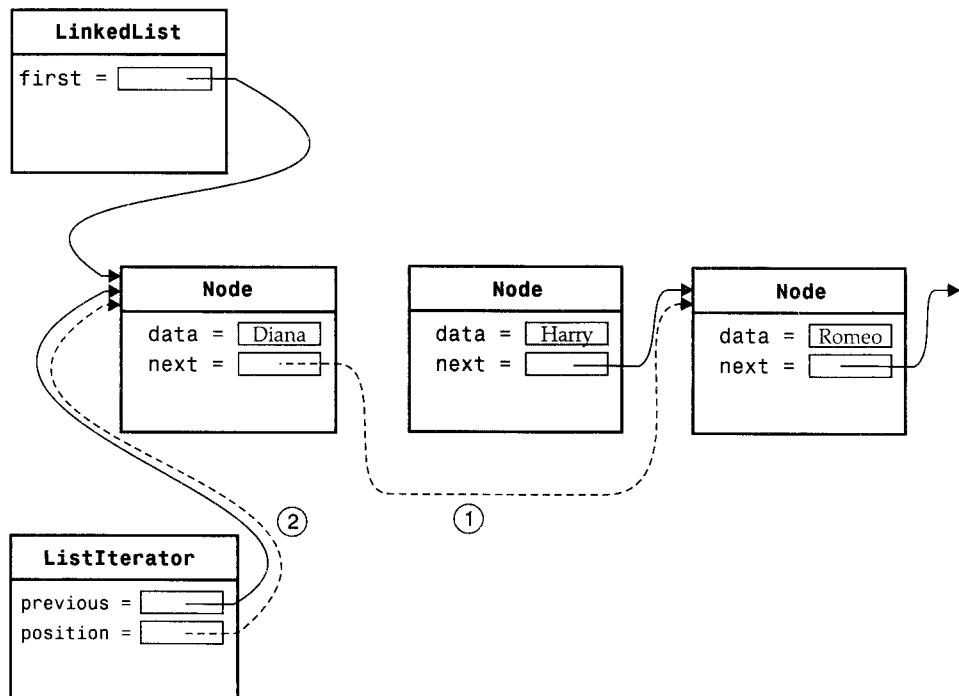
Secondo la dichiarazione del metodo `remove`, non è lecito invocare `remove` due volte di seguito: per questo motivo, il metodo `remove` rende il riferimento `previous` uguale a `position`.

**Figura 6** Prima della rimozione

Rimozione di un nodo interno a una lista concatenata.



Dopo la rimozione



```

class LinkedListIterator implements ListIterator
{
    ...
    public void remove()
    {
        if (previous == position)
            throw new IllegalStateException();
        if (position == first)
        {
            removeFirst();
        }
        else
        {
            previous.next = position.next; // 1 in figura
        }
        position = previous; // 2 in figura
    }
    ...
}

```

Infine, l'operazione più complessa è l'aggiunta di un nodo, che va inserito dopo quello visitato più recentemente dall'iteratore (osservate la Figura 7).

```

class LinkedListIterator implements ListIterator
{
    ...
    public void add(Object element)
    {
        if (position == null)
        {
            addFirst(element);
            position = first;
        }
        else
        {
            Node newNode = new Node();
            newNode.data = element;
            newNode.next = position.next; // 1 in figura
            position.next = newNode; // 2 in figura
            position = newNode; // 3 in figura
        }
        previous = position;
    }
    ...
}

```

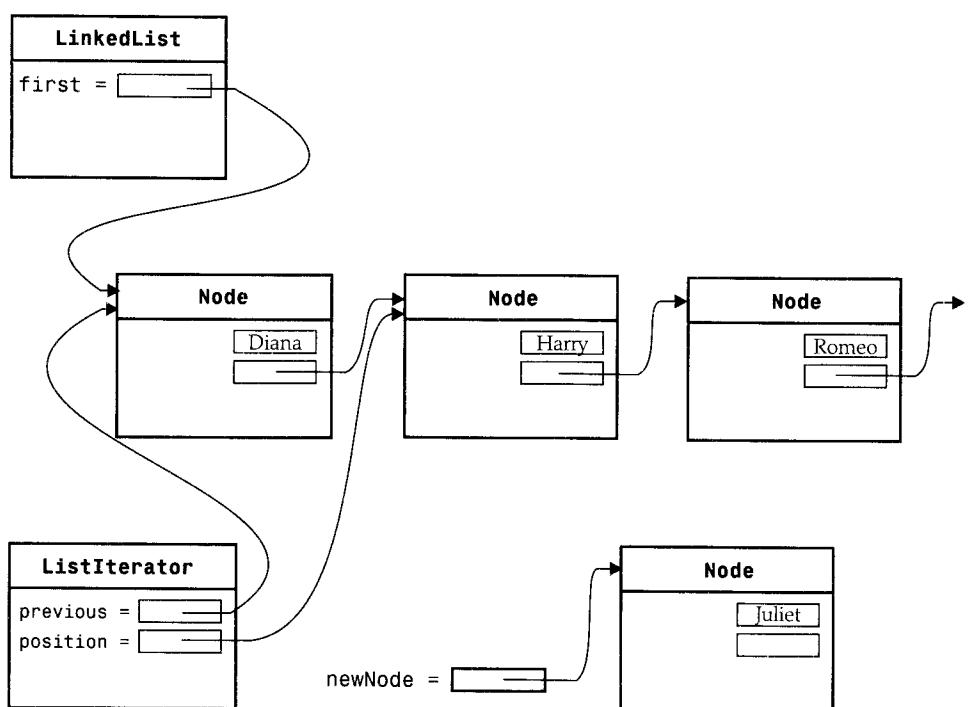
Al termine di questo paragrafo trovate la realizzazione completa della nostra classe `LinkedList`.

Adesso sapete come utilizzare la classe `LinkedList` della libreria Java e avete avuto modo di "sbirciare sotto il cofano" per vedere come vengono realizzate le liste concatenate.

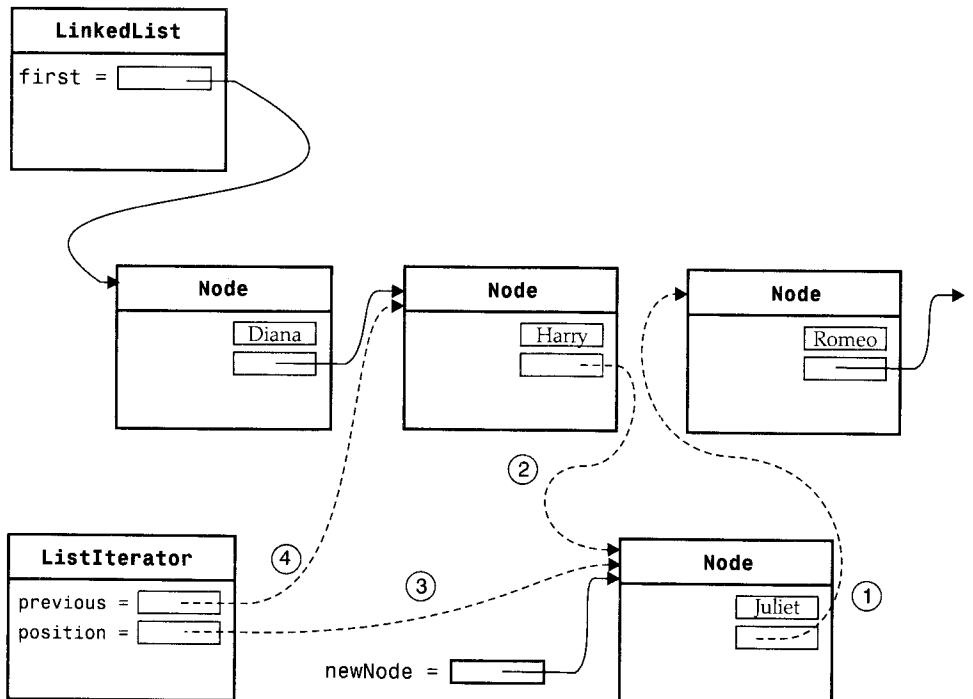
**Figura 7**

Inserimento di un nuovo nodo all'interno di una lista concatenata.

Prima dell'inserimento



Dopo l'inserimento



**File ch14/implist/LinkedList.java**

```
import java.util.NoSuchElementException;

/**
 * Una lista concatenata è una sequenza di nodi dotata di
 * efficienti meccanismi di inserimento e rimozione.
 * Questa classe ha un sottoinsieme dei metodi della
 * classe java.util.LinkedList.
 */
public class LinkedList
{
    private Node first;

    /**
     * Costruisce una lista concatenata vuota.
     */
    public LinkedList()
    {
        first = null;
    }

    /**
     * Restituisce il primo elemento della lista concatenata.
     * @return il primo elemento della lista concatenata
     */
    public Object getFirst()
    {
        if (first == null)
            throw new NoSuchElementException();
        return first.data;
    }

    /**
     * Elimina il primo elemento dalla lista concatenata.
     * @return l'elemento eliminato
     */
    public Object removeFirst()
    {
        if (first == null)
            throw new NoSuchElementException();
        Object element = first.data;
        first = first.next;
        return element;
    }

    /**
     * Aggiunge un elemento all'inizio della lista concatenata.
     * @param element l'elemento da aggiungere
     */
    public void addFirst(Object element)
    {
        Node newNode = new Node();
        newNode.data = element;
        newNode.next = first;
```

```
        first = newNode;
    }

    /**
     * Restituisce un iteratore per questa lista.
     * @return un iteratore per questa lista
    */
    public ListIterator listIterator()
    {
        return new LinkedListIterator();
    }

    class Node
    {
        public Object data;
        public Node next;
    }

    class LinkedListIterator implements ListIterator
    {
        private Node position;
        private Node previous;

        /**
         * Costruisce un iteratore che punta
         * all'inizio della lista concatenata.
        */
        public LinkedListIterator()
        {
            position = null;
            previous = null;
        }

        /**
         * Sposta l'iteratore nella posizione successiva.
         * @return l'elemento visitato
        */
        public Object next()
        {
            if (!hasNext())
                throw new NoSuchElementException();
            previous = position; // memorizza per remove

            if (position == null)
                position = first;
            else
                position = position.next;

            return position.data;
        }

        /**
         * Controlla se c'è un elemento dopo la posizione attuale.
         * @return true se c'è un elemento dopo la posizione attuale
        */
        public boolean hasNext()
```

```

    {
        if (position == null)
            return first != null;
        else
            return position.next != null;
    }

    /**
     * Aggiunge un elemento prima della posizione attuale
     * e posiziona l'iteratore dopo l'elemento inserito.
     * @param element l'elemento da aggiungere
    */
    public void add(Object element)
    {
        if (position == null)
        {
            addFirst(element);
            position = first;
        }
        else
        {
            Node newNode = new Node();
            newNode.data = element;
            newNode.next = position.next;
            position.next = newNode;
            position = newNode;
        }
        previous = position;
    }

    /**
     * Elimina l'elemento visitato più recentemente.
     * Può essere invocato soltanto dopo un'invocazione di next().
    */
    public void remove()
    {
        if (previous == position)
            throw new IllegalStateException();

        if (position == first)
        {
            removeFirst();
        }
        else
        {
            previous.next = position.next;
        }
        position = previous;
    }

    /**
     * Imposta il dato presente nell'elemento visitato
     * più recentemente.
     * @param element il dato da impostare
    */
    public void set(Object element)
}

```

```

    {
        if (position == null)
            throw new NoSuchElementException();
        position.data = element;
    }
}
}

```

### File ch14/implist/ListIterator.java

```

/*
 * Un iteratore di lista consente l'accesso a una posizione in una lista
 * concatenata. Questa interfaccia contiene un sottoinsieme dei metodi
 * dell'interfaccia java.util.ListIterator, escludendo quelli per
 * l'attraversamento all'indietro.
 */
public interface ListIterator
{
    /**
     * Sposta l'iteratore nella posizione successiva.
     * @return l'elemento visitato
    */
    Object next();

    /**
     * Controlla se c'è un elemento dopo la posizione attuale.
     * @return true se c'è un elemento dopo la posizione attuale
    */
    boolean hasNext();

    /**
     * Aggiunge un elemento prima della posizione attuale
     * e posiziona l'iteratore dopo l'elemento inserito.
     * @param element l'elemento da aggiungere
    */
    void add(Object element);

    /**
     * Elimina l'elemento visitato più recentemente.
     * Può essere invocato soltanto dopo un'invocazione di next().
    */
    void remove();

    /**
     * Imposta il dato presente nell'elemento visitato
     * più recentemente.
     * @param element il dato da impostare
    */
    void set(Object element);
}

```

### Auto-valutazione

3. Seguite passo dopo passo l'esecuzione del metodo `addFirst` durante l'inserimento di un elemento in una lista vuota.



4. L'astrazione realizzata da un iteratore prevede che la sua posizione sia situata tra due elementi, come si può vedere nella Figura 3. Il riferimento `position` punta, in realtà, all'elemento di sinistra o all'elemento di destra?
5. Perché il metodo `add` prende in esame due casi distinti?



## Argomenti avanzati 14.2

### Classi interne statiche

La prima volta che avete visto l'utilizzo delle classi interne è stato in riferimento ai gestori di eventi. Le classi interne sono utili in quel contesto perché i loro metodi hanno il privilegio di accedere alle variabili di esemplare private di oggetti della classe esterna che le contiene. Le stesse circostanze valgono per la classe interna `LinkedListIterator` nel codice dimostrativo di questo paragrafo, in quanto l'iteratore ha bisogno di accedere alla variabile di esemplare `first` della sua lista concatenata.

Tuttavia, la classe interna `Node` non ha bisogno di accedere alla classe esterna, perché non ha metodi; di conseguenza, non è necessario memorizzare all'interno di ciascun oggetto di tipo `Node` un riferimento a un esemplare della classe esterna. Per eliminare tale riferimento potete dichiarare la classe interna come `static`:

```
public class LinkedList
{
    ...
    static class Node
    {
        ...
    }
}
```

La parola chiave `static` ha, in questo contesto, lo scopo di indicare che gli oggetti della classe interna non dipendono dagli oggetti della classe esterna che li genera. In particolare, i metodi di una classe interna statica non possono accedere alle variabili di esemplare della classe esterna. Dichiarare `static` la classe interna è più efficiente, dal momento che così i suoi oggetti non memorizzano un riferimento a un esemplare della classe esterna.

La classe interna `LinkedListIterator`, invece, non può essere dichiarata statica, perché usa la variabile `first` della classe esterna `LinkedList`.

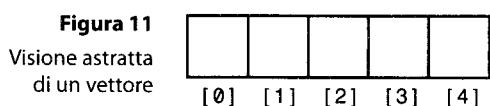
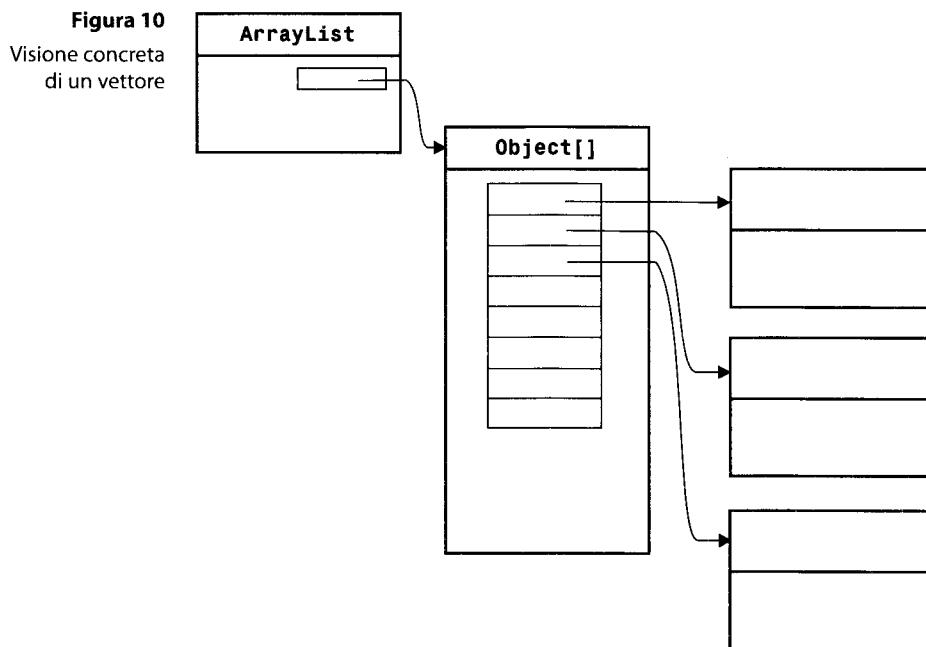
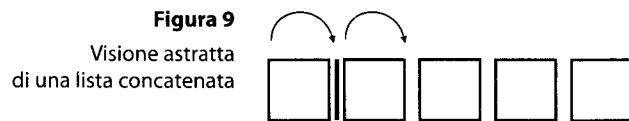
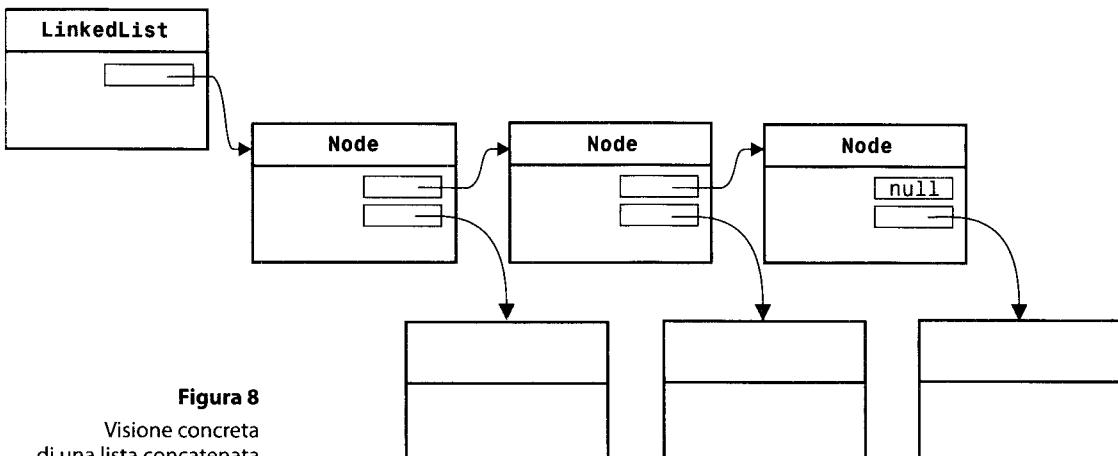
## 14.3 Tipi di dati astratti

Ci sono due diversi punti di vista dai quali osservare una lista concatenata. Uno consiste nel pensare a una realizzazione concreta di tale lista, come sequenza di nodi tra loro collegati (si veda la Figura 8).

D'altra parte, si può pensare al concetto *astratto* di lista concatenata: una lista concatenata è una sequenza ordinata di dati che può essere visitata con un iteratore (si veda la Figura 9).

Ci sono, analogamente, due modi per considerare un vettore (o lista sequenziale, *array list*). Ovviamente, un vettore ha una realizzazione concreta: un array di riferimenti a oggetti riempito solo in parte (si veda la Figura 10). Solitamente, però, quando si usa un vettore non si pensa alla sua realizzazione concreta, se ne considera l'astrazione: una

**Un tipo di dati astratto definisce le operazioni fondamentali sui dati ma non ne specifica una implementazione.**



sequenza ordinata di dati, a ciascuno dei quali si può accedere mediante un numero intero (si veda la Figura 11).

La realizzazione concreta di una lista concatenata e di un vettore sono abbastanza diverse. Le loro astrazioni, di converso, sembrano essere a prima vista simili: per apprezzarne le differenze, consideriamo le loro interfacce pubbliche, ridotte a ciò che è essenziale.

Un vettore consente l'accesso *casuale* a tutti gli elementi: basta specificare un indice intero, e si ottiene l'elemento corrispondente.

```
public class ArrayList
{
    ...
    public Object get(int index) {...}
    public void set(int index, Object element) {...}
    ...
}
```

In una lista concatenata, invece, l'accesso agli elementi è un po' più complicato. Una lista concatenata consente l'accesso *sequenziale*: occorre chiedere alla lista di fornire un iteratore. Usando quell'iteratore, possiamo scandire facilmente gli elementi della lista uno alla volta, ma se vogliamo andare a un particolare elemento, diciamo il centesimo, dobbiamo prima attraversare tutti gli elementi che lo precedono.

```
public class LinkedList
{
    ...
    public ListIterator listIterator() {...}
    ...

public interface ListIterator
{
    Object next();
    boolean hasNext();
    void add(Object element);
    void remove();
    void set(Object element);
    ...
}
```

Mostriamo qui soltanto le operazioni *fondamentali* delle liste sequenziali e delle liste concatenate, ma altre operazioni possono essere composte usando quelle fondamentali: ad esempio, potete inserire o rimuovere un elemento in un vettore spostando tutti gli elementi che si trovano al di là dell'indice in cui avviene l'inserimento o la rimozione, invocando più volte *get* e *set*.

La classe `ArrayList` ha, ovviamente, dei metodi per aggiungere ed eliminare elementi al suo interno, anche se sono lenti. Allo stesso modo, la classe `LinkedList` ha i metodi *get* e *set* che consentono l'accesso a qualsiasi elemento nella lista concatenata, anche se in modo molto inefficiente, eseguendo ripetutamente accessi sequenziali.

In realtà, il termine `ArrayList` sta a significare che chi ha realizzato la classe voleva combinare le interfacce di un array e di una lista. Anche se la cosa può creare un po' di confusione, sia la classe `ArrayList` sia la classe `LinkedList` realizzano un'interfaccia chiamata `List`, che definisce operazioni sia per l'accesso casuale sia per l'accesso sequenziale.

Questa terminologia non è comunemente utilizzata al di fuori della libreria di Java, per cui ne adottiamo una più tradizionale: chiameremo i tipi di dati astratti che abbiamo appena analizzato *array* e *lista*. La libreria di Java fornisce realizzazioni concrete per questi tipi astratti, rispettivamente `ArrayList` e `LinkedList`, mentre in altre librerie potremo trovare diverse realizzazioni concrete. Gli array del linguaggio Java sono un'altra realizzazione del tipo astratto *array*.

Per comprendere compiutamente un tipo di dati astratto, dobbiamo conoscere non soltanto le sue operazioni fondamentali, ma anche la loro efficienza relativa.

In una lista, vista come tipo di dati astratto, è possibile aggiungere o rimuovere un elemento in un tempo costante (nell'ipotesi che l'iteratore si trovi già nella posizione corretta): infatti, per aggiungere o rimuovere un nodo è necessario modificare un numero fisso di riferimenti, indipendentemente dalla dimensione della lista. Usando la notazione  $O$ -grande, un'operazione che richiede una quantità di tempo limitata indipendentemente dal numero di elementi presenti nella struttura viene indicata come  $O(1)$ . Anche l'accesso casuale in un array richiede un tempo  $O(1)$ .

Aggiungere o rimuovere un elemento qualsiasi in un array, visto come tipo di dati astratto, richiede un tempo  $O(n)$ , dove  $n$  è la dimensione dell'array, perché in media devono essere spostati  $n/2$  elementi. L'accesso casuale in una lista richiede un tempo  $O(n)$  perché devono essere visitati, in media,  $n/2$  elementi.

La Tabella 3 riporta queste informazioni per array e liste.

Perché mai dovremmo pensare ai tipi di dati astratti? Se realizzate un particolare algoritmo, potete determinare quali operazioni avete bisogno di compiere sulle strutture di dati che vengono manipolate e determinare il tipo astratto che realizza in modo efficiente tali operazioni, senza essere distratti da dettagli realizzativi.

Ad esempio, immaginate di avere una raccolta ordinata di oggetti e di voler localizzare dati al suo interno usando l'algoritmo di ricerca binaria (si veda il Paragrafo 13.7). Tale algoritmo compie un accesso casuale al centro della raccolta, seguito da altri accessi casuali: perché l'algoritmo sia efficiente, è quindi essenziale che l'accesso casuale sia veloce. Sapendo che, in astratto, un array fornisce un accesso casuale veloce mentre la lista non lo fa, cercate realizzazioni concrete del tipo astratto *array*: non farete la pazzia di usare un oggetto di tipo `LinkedList`, anche se tale classe fornisce i metodi `get` e `set`.

Nel prossimo paragrafo vedrete altri esempi di tipi di dati astratti.

**Tabella 3**

Efficienza delle operazioni per i tipi di dati astratti *array* e *lista*

| Operazione                        | Array  | Lista  |
|-----------------------------------|--------|--------|
| Accesso casuale                   | $O(1)$ | $O(n)$ |
| Passo di attraversamento lineare  | $O(1)$ | $O(1)$ |
| Aggiunta/rimozione di un elemento | $O(n)$ | $O(1)$ |

## Auto-valutazione

6. Qual è il vantaggio della visione astratta dei tipi di dati?
7. Come delineereste una visione astratta di una lista doppiamente concatenata? È una sua visione concreta?
8. In confronto all'algoritmo di ricerca lineare, quanto è più lento l'algoritmo di ricerca binaria eseguito su una lista?





## Note di cronaca 14.1

### Standardizzazione

Ogni giorno vi confrontate con i vantaggi della standardizzazione. Quando comprate una lampadina, potete essere certi che si avvierà correttamente nel portalamppada senza averlo misurato a casa e senza misurare la lampadina nel negozio. Al contrario, potete aver conosciuto la gravità 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 e averne ottenuto i file di classi, potete eseguirli su 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 le macchine virtuali non si comportassero tutte 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 programmazione 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 volontari, come Internet Engineering Task Force (IETF) e World Wide Web

Consortium (W3C). Sul sito di IETF potete trovare i documenti RFC (Request for Comment, <http://www.ietf.org/rfc.html>) che rendono standard molti protocolli di Internet: ad esempio, RFC 822 riguarda il formato della posta elettronica, mentre RFC 2616 definisce il protocollo HTTP (Hypertext Transmission Protocol) che viene usato per fornire pagine Web ai browser. Il consorzio W3C (<http://www.w3c.org>) si occupa del linguaggio HTML (Hypertext Markup Language), 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 per gli standard, 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.

Quando un'azienda inventa una nuova tecnologia, ha tutto interesse che la sua invenzione diventi uno standard, in modo che altre aziende producano strumenti che possano lavorare con essa, incrementandone così le probabilità di successo. D'altra parte, mettendo l'invenzione nelle mani di un comitato di standardizzazione, specialmente di uno che voglia garantire una procedura equa,

l'azienda può perdere il controllo dello standard. Per tale ragione, lo standard di Java viene sviluppato da Java Community Process, un consorzio di industrie controllato da Sun Microsystems, che ha inventato Java.

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 professionisti, 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 la classe `LinkedList` della libreria standard Java; tuttavia, molti informatici questa classe non la piace, perché l'interfaccia confonde la distinzione tra i tipi astratti lista e array, oltre al fatto che i suoi iteratori sono scomodi da usare. Dovreste usare nel vostro codice la classe `LinkedList`, o sarebbe meglio realizzare una lista migliore? Se scegliete la prima opportunità, dovete fare i conti con una realizzazione non ottimale. Nel secondo caso, altri programmati potrebbero avere difficoltà nella comprensione del vostro codice, perché non hanno familiarità con la vostra classe lista.

## 14.4 Pile e code

In questo paragrafo prenderemo in considerazione due tipi di dati astratti molto comuni che consentono l'inserimento e la rimozione di oggetti soltanto alle estremità, ma non in posizioni intermedie. Una *pila* (*stack*) consente l'inserimento e la rimozione di elementi a una sola estremità, che viene tradizionalmente chiamata *cima* (*top*) della pila. Per rappresentare concettualmente una pila, pensate a una pila di libri.

**Una pila è una raccolta di oggetti che vengono estratti con modalità "last in, first out".**

Nuovi oggetti vengono inseriti in cima alla pila e le rimozioni avvengono 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 e poi li estraete, otterrete C, B e A. Le operazioni di inserimento e di estrazione vengono tradizionalmente chiamate *push* e *pop*.

**Una coda è una raccolta di oggetti che vengono estratti con modalità "first in, first out".**

Una *coda* (*queue*) è simile a una pila, ma si inseriscono elementi a un'estremità della coda (la “fine”, *tail*) e li si rimuovono dall'altra estremità (la “testa”, *head*). Per visualizzare il concetto di coda, pensate semplicemente a una coda di persone in attesa: le persone si aggiungono alla fine della coda e aspettano finché non hanno raggiunto la testa della coda stessa. Le code conservano gli elementi in modo che “il primo entrato sia il primo a uscire” (“*first in, first out*”, modalità FIFO): gli oggetti vengono quindi rimossi nello stesso ordine in cui sono stati inseriti.

Ci sono molti possibili utilizzi di code e pile nell'informatica. Ad esempio, il sistema che gestisce l'interfaccia utente grafica in Java conserva in una coda tutti gli eventi relativi al mouse e alla tastiera: gli eventi vengono inseriti nella coda quando il sistema operativo li rende noti all'applicazione, la quale li rimuove dalla coda nello stesso ordine in cui erano stati inseriti e li trasferisce al gestore di eventi appropriato. Un altro esempio è una 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 accoda tale file nella coda di stampa. Quando la stampante ha terminato la stampa di un file, estrae il successivo dalla coda, per cui i lavori vengono stampati usando la regola “il primo entrato è il primo a uscire”, che è una strategia accettabile per gli utenti della stampante condivisa.

Le pile vengono usate, invece, quando si deve realizzare una strategia di tipo “l'ultimo entrato è il primo a uscire”. Ad esempio, considerate un algoritmo che cerchi di individuare un percorso per uscire da un labirinto. Quando l'algoritmo incontra un incrocio, inserisce tale posizione in cima a una pila ed esplora la prima diramazione che parte dall'incrocio: se si tratta di un percorso a fondo chiuso, torna alla posizione che si trova in cima alla pila; se tutti i percorsi che si dipartono da tale posizione sono a fondo chiuso, la posizione stessa viene eliminata dalla cima della pila, rendendo visibile e attiva la posizione dell'incrocio precedente. 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 una pila i suoi parametri e le sue variabili locali; quando il metodo termina, si estraggono dalla pila parametri e variabili. Tale pila rende possibili le invocazioni ricorsive.

Nella libreria di Java è presente una classe `Stack` che realizza il tipo astratto pila e le operazioni `push` e `pop`.

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 libreria standard mette a disposizione un certo numero di classi che realizzano code, pensate per programmi nei quali vengono eseguite in parallelo più attività, chiamate *thread*, di cui in questo libro non parliamo. Anche la classe `LinkedList` realizza l'interfaccia `Queue`, per cui, quando vi serve una cosa, potete usare quella:

```
Queue<String> q = new LinkedList<String>;
```

La Tabella 4 mostra come si usano, in Java, i metodi della pila e della coda.

La classe `Stack` della libreria Java usa un vettore per realizzare una pila; l'Esercizio P14.15 mostra come usare, invece, una lista concatenata.

Non dovreste assolutamente usare un vettore per realizzare una coda: rimuovere il primo elemento di un vettore è inefficiente, perché tutti gli altri elementi devono essere spostati verso l'inizio. Una coda si realizza in modo efficiente usando una lista concatenata. Tuttavia, l'Esercizio P14.16 mostra come realizzare una coda in modo efficiente usando un array “circolare”, nel quale tutti gli elementi rimangono nella stessa posizione in cui sono stati inseriti: quando si aggiungono o rimuovono elementi, vengono modificati soltanto i valori degli indici che denotano la testa e la fine della coda.

In questo capitolo avete visto i due tipi di dati astratti più basilari, array e liste, oltre a loro realizzazioni concrete. Avete anche studiato i tipi pila e coda: nel prossimo capitolo vedrete altri tipi di dati che richiedono tecniche di realizzazione più sofisticate.

**Tabella 4**

Lavorare con code e pile

```
Queue<Integer> q = new LinkedList<Integer>();
q.add(1); q.add(2); q.add(3);
int head = q.remove();

head = q.peek();

Stack<Integer> s = new Stack<Integer>();
s.push(1); s.push(2); s.push(3);
int top = s.pop();
head = s.peak();
```

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| La classe <code>LinkedList</code> realizza l'interfaccia <code>Queue</code> .<br>Inserimenti alla fine della coda; ora <code>q</code> contiene [1, 2, 3].<br>Eliminazione dall'inizio della coda; ora <code>head</code> vale 1 e <code>q</code> contiene [2, 3].<br>Ispezione dell'elemento presente all'inizio della coda, senza rimuoverlo; ora <code>head</code> vale 2.<br>Costruisce una pila vuota.<br>Inserimenti in cima alla pila; ora <code>s</code> contiene [1, 2, 3].<br>Eliminazione dalla cima della pila; ora <code>top</code> vale 3 e <code>s</code> contiene [1, 2].<br>Ispezione dell'elemento presente in cima alla pila, senza rimuoverlo; ora <code>head</code> vale 2. |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|



## Auto-valutazione

9. Tracciate uno schema, simile alle Figure 9 e 11, relativo al tipo di dati astratto “coda”.
10. Perché una pila non è adatta alla gestione di una stampante?

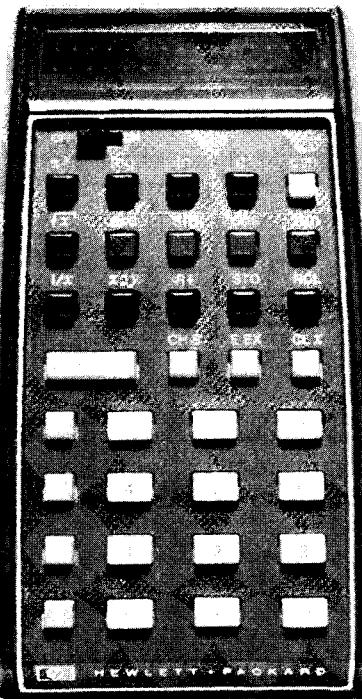


## Note di cronaca 14.2

### 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*).

La notazione RPN vi potrà apparire strana, ma si tratta solo di un caso dovuto agli avvenimenti storici. Se i primi matematici ne



avessero compreso i vantaggi, 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.

## Esempi completi 14.1

### Una calcolatrice in notazione polacca inversa (RPN)

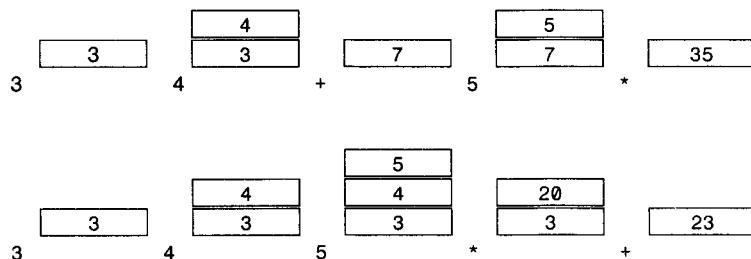
Nella normale notazione algebrica, gli operatori aritmetici sono posizionati fra i relativi operandi, come nell'espressione  $3 + 4$ . Come visto nelle Note di cronaca 14.2, invece, usando la notazione polacca inversa (RPN) si scrivono gli operatori *dopo* i relativi argomenti:  $3 \ 4 \ +$ . La tabella di Note di cronaca 14.2 mostra alcuni esempi e, come potete vedere, con la notazione RPN non c'è mai bisogno di usare parentesi.

La valutazione di espressioni che usano RPN è semplice se si dispone di una pila: ogni operando viene inserito nella pila; ogni operatore estrae dalla pila i propri operandi, esegue l'operazione e inserisce il risultato nella pila. Ad esempio, l'espressione  $3 \ 4 \ +$  viene valutata in tre passi:



- Inserisco 3 nella pila.
- Inserisco 4 nella pila.
- Estraggo dalla pila i due valori più in alto, applico l'operatore + e inserisco nella pila il risultato, 7.

La figura mostra in che ordine vengono eseguiti i calcoli per valutare le due espressioni  $3 \ 4 \ + \ 5 \ *$  e  $3 \ 4 \ 5 \ * \ +$ .



Per realizzare una calcolatrice che operi in questo modo, usiamo una variabile per memorizzare la pila (ricordate che non possiamo usare `Stack<int>` perché i tipi primitivi non possono essere usati come tipi parametrici):

```
Stack<Integer> results;
```

Per progettare l'interfaccia utente abbiamo diverse alternative. Potremmo chiedere all'utente di fornire l'intera espressione in un'unica riga, visualizzando poi il risultato finale della valutazione, ad esempio così:

```
Please enter the expression: 3 4 + 5 *
Result: 35
```

Si tratta di un'interfaccia che appare piuttosto naturale se l'utente è interessato solamente al risultato dell'espressione, ma, se vogliamo illustrare il meccanismo interno di funzionamento della calcolatrice, è bene mostrare il contenuto della pila dopo ogni azione. Di conseguenza, decidiamo di chiedere all'utente di fornire un operando o un operatore per riga.

Quando leggiamo un operatore, estraiamo i suoi operandi dalla pila, eseguiamo l'operazione e inseriamo nella pila il risultato ottenuto. Ecco, ad esempio, il codice relativo all'operatore +:

```
if (input.equals("+"))
{
    results.push(results.pop() + results.pop());
}
```

Con gli operatori - e /, però, dobbiamo fare più attenzione. Esaminiamo l'espressione  $3 \ 4 \ -$ . Dopo aver elaborato in ingresso i numeri 3 e 4, la pila contiene il valore 4 in cima. Se calcolassimo, semplicemente

```
results.pop() - results.pop()
```

otterremmo la differenza 4 - 3. Il codice che segue calcola la differenza nell'ordine corretto:

```
if (input.equals("-"))
{
    Integer arg2 = results.pop();
    results.push(results.pop() - arg2);
}
```

Ecco il programma completo per la simulazione di una calcolatrice in notazione polacca inversa.

### File ch14/calc/Calculator.java

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

/**
 * Questa calcolatrice usa la notazione polacca inversa.
 */
public class Calculator
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        Stack<Integer> results = new Stack<Integer>();
        System.out.println("Enter one number or operator per line, Q to quit.");
        boolean done = false;
        while (!done)
        {
            String input = in.nextLine();

            // se è un operatore, estrai gli operandi e impila il risultato

            if (input.equals("+"))
            {
                results.push(results.pop() + results.pop());
            }
            else if (input.equals("-"))
            {
                Integer arg2 = results.pop();
                results.push(results.pop() - arg2);
            }
            else if (input.equals("*") || input.equals("x"))
            {
                results.push(results.pop() * results.pop());
            }
            else if (input.equals("/"))
            {
                Integer arg2 = results.pop();
                results.push(results.pop() / arg2);
            }
            else if (input.equals("Q") || input.equals("q"))
            {
```

```
        done = true;
    }
} else
{
    // non è un operatore, impila il valore
    results.push(Integer.parseInt(input));
}
System.out.println(results);
}
}
```

## Esecuzione del programma

```
Enter one number or operator per line, Q to quit.  
3  
[3]  
4  
[3, 4]  
+  
[7]  
5  
[7, 5]  
*  
[35]  
Q
```

# Riepilogo degli obiettivi di apprendimento

## **Lista concatenata: descrizione e uso degli iteratori**

- 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 lista concatenata è un'operazione efficiente, anche in posizioni intermedie.
  - Visitare in sequenza gli elementi di una lista concatenata è efficiente, ma accedervi in ordine casuale non lo è.
  - Per accedere agli elementi presenti in una lista concatenata si usa un iteratore o (cursore).

## Realizzazione di liste concatenate

- Un esemplare di lista concatenata memorizza un riferimento al primo nodo, mentre ciascun nodo memorizza un riferimento al nodo successivo.
  - Un oggetto iteratore memorizza un riferimento all'ultimo nodo che ha visitato nella lista.
  - La realizzazione di operazioni che modificano una lista concatenata è una sfida: bisogna verificare con cura che vengano aggiornati in modo corretto tutti i riferimenti ai nodi.

**La nozione di tipo di dato astratto e il comportamento astratto di array, liste, pile e code**

- Un tipo di dati astratto definisce le operazioni fondamentali sui dati ma non ne specifica una implementazione.

- Una lista, vista come tipo di dati astratto, è una sequenza ordinata di elementi che può essere visitata con accesso sequenziale e che consente l'inserimento e la rimozione di elementi in qualsiasi posizione in un tempo  $O(1)$ .
- Un array, visto come tipo di dati astratto, è una sequenza ordinata di elementi che consente l'accesso casuale in un tempo  $O(1)$  mediante un indice intero.
- Una pila è una raccolta di oggetti che vengono estratti con modalità “last in, first out”.
- Una coda è una raccolta di oggetti che vengono estratti con modalità “first in, first out”.

## Classi, oggetti e metodi presentati nel capitolo

|                                            |                                              |
|--------------------------------------------|----------------------------------------------|
| <code>java.util.Collection&lt;E&gt;</code> | <code>addLast</code>                         |
| <code>add</code>                           | <code>getFirst</code>                        |
| <code>contains</code>                      | <code>getLast</code>                         |
| <code>iterator</code>                      | <code>removeFirst</code>                     |
| <code>remove</code>                        | <code>removeLast</code>                      |
| <code>size</code>                          | <code>java.util.List&lt;E&gt;</code>         |
| <code>java.util.Iterator&lt;E&gt;</code>   | <code>listIterator</code>                    |
| <code>hasNext</code>                       | <code>java.util.ListIterator&lt;E&gt;</code> |
| <code>next</code>                          | <code>add</code>                             |
| <code>remove</code>                        | <code>hasPrevious</code>                     |
| <code>java.util.LinkedList&lt;E&gt;</code> | <code>previous</code>                        |
| <code>addFirst</code>                      | <code>set</code>                             |

## Esercizi di ripasso

- \* **Esercizio R14.1.** Spiegate che cosa viene visualizzato dal codice seguente. Tracciate uno schema della lista concatenata dopo ciascun passo. Disegnate soltanto i collegamenti in avanti, come nella Figura 1.

```
LinkedList<String> staff = new LinkedList<String>();
staff.addFirst("Harry");
staff.addFirst("Diana");
staff.addFirst("Tom");
System.out.println(staff.removeFirst());
System.out.println(staff.removeFirst());
System.out.println(staff.removeFirst());
```

- \* **Esercizio R14.2.** Spiegate che cosa viene visualizzato dal codice seguente. Tracciate uno schema della lista concatenata dopo ciascun passo. Disegnate soltanto i collegamenti in avanti, come nella Figura 1.

```
LinkedList<String> staff = new LinkedList<String>();
staff.addFirst("Harry");
staff.addFirst("Diana");
staff.addFirst("Tom");
System.out.println(staff.removeLast());
System.out.println(staff.removeFirst());
System.out.println(staff.removeLast());
```

- \* **Esercizio R14.3.** Spiegate che cosa viene visualizzato dal codice seguente. Tracciate uno schema della lista concatenata dopo ciascun passo. Disegnate soltanto i collegamenti in avanti, come nella Figura 1.

```
LinkedList<String> staff = new LinkedList<String>();
staff.addFirst("Harry");
staff.addLast("Diana");
staff.addFirst("Tom");
System.out.println(staff.removeLast());
System.out.println(staff.removeFirst());
System.out.println(staff.removeLast());
```

- \* **Esercizio R14.4.** 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<String>();
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());
```

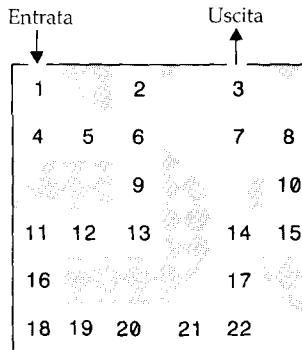
- \* **Esercizio R14.5.** Spiegate che cosa viene visualizzato dal codice seguente. Tracciate uno schema della lista concatenata e la posizione dell'iteratore dopo ciascun passo.

```
LinkedList<String> staff = new LinkedList<String>();
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());
```

- \*\* **Esercizio R14.6.** La classe lista concatenata presente nella libreria di Java consente le operazioni `addLast` e `removeLast`. Per eseguire queste operazioni in modo efficiente, la classe `LinkedList` ha un riferimento aggiuntivo, `last`, all'ultimo nodo della lista concatenata. Tracciate un diagramma "prima/dopo" delle modifiche dei collegamenti in una lista concatenata per effetto dei metodi `addLast` e `removeLast`.
- \*\* **Esercizio R14.7.** La classe lista concatenata presente nella libreria di Java consente gli iteratori bidirezionali. Per andare all'indietro in modo efficiente, ciascun oggetto di tipo `Node` ha un ulteriore riferimento, `previous`, al nodo precedente nella lista concatenata. Tracciate un diagramma "prima/dopo" delle modifiche dei collegamenti in una lista concatenata per effetto dei metodi

`addFirst` e `removeFirst` che mostri in che modo devono essere aggiornati i collegamenti `previous`.

- ★★ **Esercizio R14.8.** Quali vantaggi e svantaggi hanno le liste rispetto agli array?
- ★★ **Esercizio R14.9.** 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 10 000 numeri di telefono e prevedete che l'elenco venga consultato diverse centinaia di volte al giorno. Per memorizzare le informazioni usereste un array o una lista?
- ★★ **Esercizio R14.10.** Supponete di dover gestire un elenco di appuntamenti. Usereste una lista o un array di oggetti di tipo `Appointment`?
- \* **Esercizio R14.11.** 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. Memorizzereste le carte in una pila o in una coda?
- \* **Esercizio R14.12.** 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?
- \* **Esercizio R14.13.** Considerate l'attraversamento di questo labirinto:



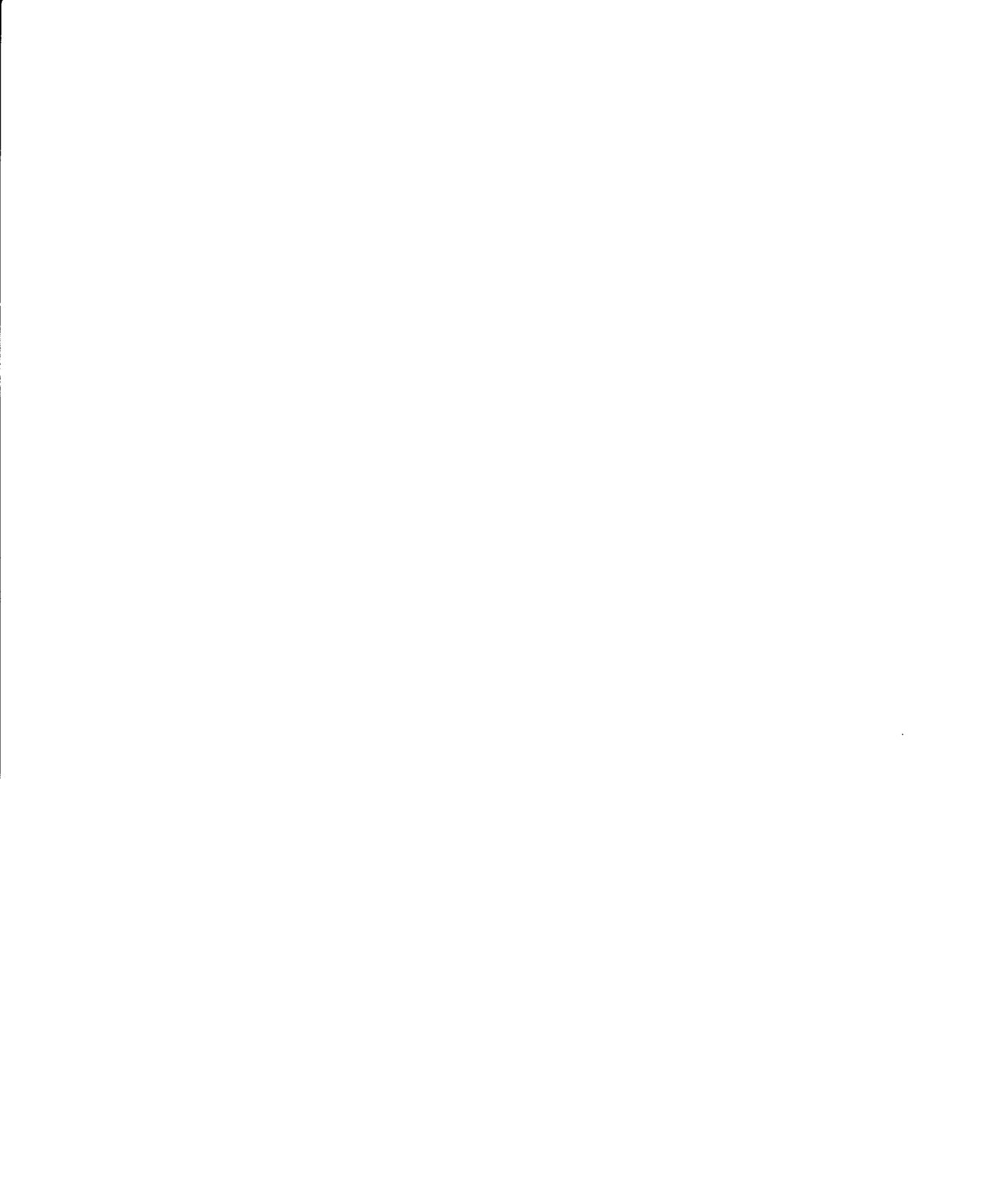
usando l'algoritmo qui descritto. La cella d'entrata diventa la cella attuale. Eseguite queste azioni, quindi ripetete:

- Se la cella attuale è adiacente all'uscita, l'algoritmo termina.
- La cella attuale viene contrassegnata come visitata.
- Le celle adiacenti in direzione nord, est, sud e ovest, che non siano state visitate, vengono inserite in una coda.
- L'elemento che si trova all'inizio della coda viene rimosso e diventa la cella attuale.

In quale ordine verranno visitate le celle del labirinto presentato come esempio?

- \* **Esercizio R14.14.** Ripetete l'esercizio precedente, usando una pila invece di una coda.

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



# 15

## Strutture di dati avanzate

### Obiettivi del capitolo

- Apprendere i tipi di dati insieme e mappa
- Capire la realizzazione di tabelle *hash*
- Saper programmare funzioni di hash
- Imparare gli alberi binari
- Acquisire familiarità con la struttura dati *heap*
- Imparare a realizzare una coda prioritaria
- Capire come si possano usare heap per effettuare ordinamenti

In questo capitolo studiamo strutture di dati più complesse rispetto a liste o array. Queste strutture assumono il pieno controllo dell'organizzazione dei propri elementi, piuttosto che lasciarli in una posizione fissa, e sono in grado di offrire migliori prestazioni per l'inserimento, la rimozione e la ricerca di elementi.

Conoscerete i tipi di dati astratti mappa e insieme, per proseguire con le realizzazioni offerte dalla libreria standard per questi tipi astratti. Vedrete come si possano realizzare efficientemente questi tipi astratti in due modi sostanzialmente diversi, usando tabelle hash o alberi.

## 15.1 Insiemi

Nel capitolo precedente avete visto due importanti strutture per dati, le liste e gli array, che hanno una caratteristica in comune: conservano gli elementi nello stesso ordine in cui vi vengono inseriti. Tuttavia, in realtà, per molte applicazioni non interessa l'ordine degli elementi presenti in un contenitore. Ad esempio, un computer può utilizzare una raccolta di oggetti che rappresentano le stampanti disponibili e il loro ordine non interessa per niente.

**Un insieme è una raccolta non ordinata di elementi distinti.**

**Gli elementi vi possono essere aggiunti, rimossi e cercati.**

In matematica tale raccolta non ordinata viene detta *insieme*. Probabilmente avrete appreso alcuni elementi di teoria degli insiemi in un corso di matematica e saprete che gli insiemi sono una nozione matematica fondamentale.

Ma quale significato ha questa parola per le strutture dati? Se una struttura non ha più la responsabilità di ricordare l'ordine in cui vengono inseriti gli elementi, è in grado di fornire migliori prestazioni per alcune delle sue operazioni? La risposta è affermativa, come vedrete più avanti in questo capitolo.

Ecco le operazioni fondamentali per un insieme:

- Aggiunta di un elemento
- Rimozione di un elemento
- Verifica di appartenenza (un particolare oggetto appartiene all'insieme?)
- Elencazione di tutti gli elementi (non necessariamente nell'ordine in cui sono stati aggiunti)

**Gli insiemi non hanno elementi duplicati. L'aggiunta di un elemento già presente viene ignorata silenziosamente.**

In matematica un insieme non può contenere elementi duplicati: se un oggetto è già presente nell'insieme, il tentativo di aggiungerlo di nuovo viene ignorato. Questo comportamento è utile anche in molte situazioni di programmazione: ad esempio, se stiamo gestendo un insieme delle stampanti disponibili, ciascuna stampante dovrebbe figurare al massimo una sola volta nell'insieme. Quindi, daremo alle operazioni `add` e `remove` degli insiemi lo stesso significato che hanno in matematica: l'aggiunta di un elemento non ha alcun effetto se l'elemento si trova già nell'insieme, mentre la rimozione di un elemento che non si trova nell'insieme viene ignorata silenziosamente.

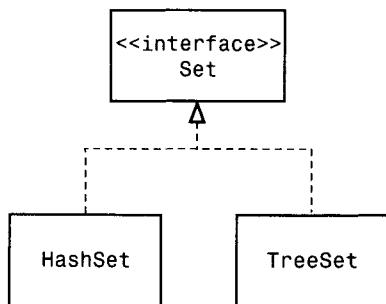
Potremmo ovviamente usare una lista concatenata per realizzare un insieme, ma l'aggiunta, la rimozione e la verifica di appartenenza sarebbero operazioni  $O(n)$ , perché dovrebbero effettuare una ricerca lineare attraverso la lista (anche per aggiungere un elemento bisogna scandire tutta la lista, per accertarsi che non si stia aggiungendo un duplicato). Come vedrete più avanti in questo capitolo, esistono strutture che possono gestire queste operazioni molto più velocemente.

Esistono, a dire il vero, due diverse strutture dati che realizzano tale obiettivo, chiamate *tabelle hash* e *alberi*. La libreria standard di Java fornisce realizzazioni di insiemi basate su entrambe tali strutture: rispettivamente, `HashSet` e `TreeSet`, che implementano entrambe l'interfaccia `Set` (si veda la Figura 1).

Se volete usare un insieme in un vostro programma, dovete scegliere una di queste implementazioni. Per usare un insieme di tipo `HashSet`, gli elementi che vi verranno inseriti devono avere il metodo `hashCode`, di cui parleremo nei Paragrafi 15.3 e 15.4 e che sono presenti in molte classi della libreria standard, tra cui `String`, `Integer`, `Point`, `Rectangle`, `Color` e tutte le classi di tipo contenitore. Di conseguenza, potete

**Le classi `HashSet` e `TreeSet` realizzano l'interfaccia `Set`.**

**Figura 1**  
Classi e interfacce  
per gli insiemi  
nella libreria standard



certamente costruire oggetti di tipo `HashSet<String>`, `HashSet<Integer>` o anche `HashSet<HashSet<Integer>>`.

La classe `TreeSet` usa una strategia differente per disporre gli elementi al proprio interno: li tiene ordinati, per cui devono essere esemplari di una classe che realizza l'interfaccia `Comparable` (vista nel Paragrafo 13.8). Le classi `String` e `Integer` soddisfano questo requisito, ma molte altre classi no; in alternativa, potete costruire un insieme di tipo `TreeSet` fornendo un oggetto di tipo `Comparator` (visto in Argomenti avanzati 13.5).

Come regola di base, usate un insieme di tipo `HashSet`, a meno che non vogliate visitare gli elementi dell'insieme in ordine.

Vediamo ora come si usa un insieme di stringhe. Per prima cosa, lo costruiamo, così

```
Set<String> names = new HashSet<String>();
```

oppure così

```
Set<String> names = new TreeSet<String>();
```

Si noti che memorizziamo il riferimento all'oggetto di tipo `HashSet<String>` o `TreeSet<String>` in una variabile di tipo `Set<String>`: dopo aver costruito l'oggetto contenitore, non importa più sapere quale realizzazione dell'insieme venga usata, ci serve solo conoscere l'interfaccia.

Aggiungere e rimuovere elementi dell'insieme è semplice:

```
names.add("Romeo");
names.remove("Juliet");
```

Il metodo `contains` verifica se un elemento appartiene all'insieme:

```
if (names.contains("Juliet")) ...
```

Per elencare tutti gli elementi di un insieme si usa un iteratore.

Infine, per elencare tutti gli elementi presenti nell'insieme, occorre un iteratore. Come per gli iteratori di lista, 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();
```

```
    Fa qualcosa con name
}
```

Oppure, invece di usare esplicitamente un iteratore, si può usare un ciclo `for` esteso, come con array e liste:

```
for(String name : names)
{
    Fa qualcosa con name
}
```

**Un iteratore visita gli elementi di un HashSet in ordine apparentemente casuale, mentre visita ordinatamente gli elementi di un TreeSet.**

**Non si può aggiungere un elemento in una particolare posizione di un iteratore di un insieme.**

Si noti che gli elementi *non* vengono visitati nell'ordine in cui sono stati inseriti: se usate un insieme di tipo `HashSet`, gli elementi vengono visitati in ordine apparentemente casuale (per un motivo che verrà spiegato nel Paragrafo 15.5), mentre se usate un insieme di tipo `TreeSet` gli elementi vengono visitati secondo l'ordine indotto dal metodo di confronto.

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. Un `ListIterator` ha un metodo `add` per aggiungere un elemento nella posizione in cui si trova l'iteratore, mentre l'interfaccia `Iterator` non ha tale metodo: non ha senso aggiungere un elemento in una particolare posizione di un insieme, perché l'insieme stesso può ordinare i propri elementi come preferisce. Quindi, si aggiunge sempre un elemento direttamente all'insieme, mai a un iteratore dell'insieme.

Tuttavia, è possibile eliminare l'elemento dell'insieme 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".

Il seguente programma di prova mostra un'applicazione pratica degli insiemi. Leggiamo tutte le parole di un dizionario, contenuto in un file, e le inseriamo in un insieme; poi, leggiamo tutte le parole di un documento, inserendole in un secondo insieme (in questo caso, il libro "Alice in Wonderland", *Alice nel paese delle meraviglie*); infine, visualizziamo 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 male (come potrete vedere, usiamo un dizionario americano e parole inglese, come *clamour*, vengono segnalate come potenziali errori).

### File ch15/spellcheck/SpellCheck.java

```
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;
import java.io.File;
import java.io.FileNotFoundException;

/**
 * Questo programma verifica quali parole di un file non
 * sono presenti in un dizionario.
 */
public class SpellCheck
```

```

{
    public static void main(String[] args)
        throws FileNotFoundException
    {
        // Legge il dizionario e il documento

        Set<String> dictionaryWords = readWords("words");
        Set<String> documentWords = readWords("alice30.txt");

        // Visualizza tutte le parole che sono presenti nel documento
        // ma non nel dizionario

        for (String word : documentWords)
        {
            if (!dictionaryWords.contains(word))
            {
                System.out.println(word);
            }
        }
    }

    /**
     * Legge tutte le parole contenute in un file.
     * @param filename il nome del file
     * @return un insieme contenente tutte le parole presenti nel
             file, convertite in minuscolo; una parola è una
             sequenza di lettere, maiuscole o minuscole.
    */
    public static Set<String> readWords(String filename)
        throws FileNotFoundException
    {
        Set<String> words = new HashSet<String>();
        Scanner in = new Scanner(new File(filename));
        // Usa come delimitatore qualunque carattere che non
        // appartiene all'intervallo a-z né all'intervallo A-Z
        in.useDelimiter("[^a-zA-Z]+");
        while (in.hasNext())
        {
            words.add(in.next().toLowerCase());
        }
        return words;
    }
}

```

## Esecuzione del programma

```

neighbouring
croqueted
pennyworth
dutchess
comfits
xii
dinn
clamour
...

```



## Auto-valutazione

1. 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?
2. Perché gli iteratori per insiemi sono diversi dagli iteratori per liste?
3. Se nel programma SpellCheck utilizzate, per la lettura del file `alice30.txt`, un insieme di tipo `TreeSet` anziché di tipo `HashSet`, vi saranno modifiche nei risultati visualizzati?
4. Quando preferirete usare un insieme di tipo `TreeSet` anziché uno di tipo `HashSet`?



## Consigli per la qualità 15.1

### 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<String>();
```

In questo modo, se decidete di usare, invece, un oggetto di tipo `TreeSet`, dovete modificare soltanto una linea.

In aggiunta a ciò, i metodi che agiscono su insiemi dovrebbero specificare parametri di tipo `Set`:

```
public static void print(Set<String> s)
```

Il metodo può così essere utilizzato con tutte le diverse realizzazioni di insiemi.

In teoria dovremmo fare la stessa raccomandazione per le liste concatenate, in particolare suggerendo di memorizzare riferimenti a `LinkedList` in variabili di tipo `List`. Tuttavia, nella libreria di Java l'interfaccia `List` è comune sia alla classe `ArrayList` sia alla classe `LinkedList`, e 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 l'accesso casuale è efficiente oppure no. Ciò è ovviamente un serio errore di progettazione della libreria standard e per tale motivo non posso raccomandare l'uso dell'interfaccia `List`. Per capire quanto sia imbarazzante quell'errore, date un'occhiata al codice sorgente per il metodo `binarySearch` della classe `Collections`: quel metodo riceve un parametro di tipo `List`, ma la ricerca binaria non ha senso per una lista concatenata, per cui il codice cerca confusamente di scoprire se la lista sia una lista concatenata, per potere, nel caso, ricorrere a una ricerca lineare!

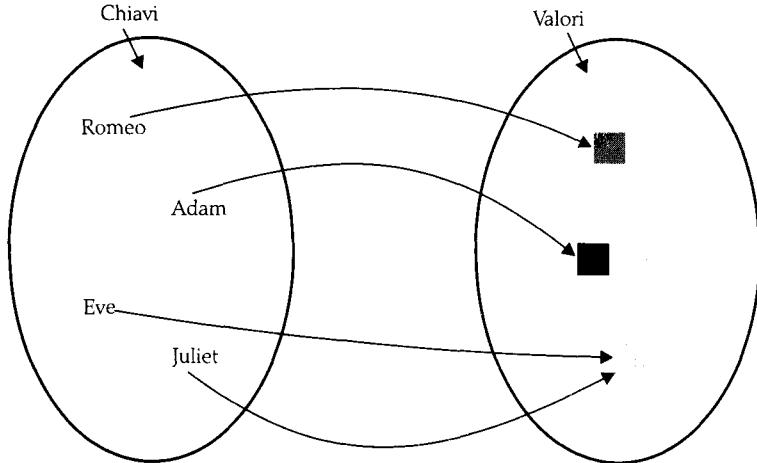
L'interfaccia `Set` e l'interfaccia `Map`, che sarà presentata nel prossimo paragrafo, sono state progettate bene e dovreste usarle.

## 15.2 Mappe

Una mappa memorizza associazioni tra oggetti che fungono da chiave e oggetti che rappresentano un valore.

Una mappa è un tipo di dato che memorizza associazioni tra *chiavi (key)* e *valori (value)*. La Figura 2 mostra un esempio tipico: una mappa che associa nomi a tonalità di grigio (potrebbe descrivere la tonalità di grigio preferito di varie persone).

**Figura 2**  
Una mappa



Parlando in termini matematici, una mappa è una funzione da un insieme, l'*insieme delle chiavi*, a un altro insieme, l'*insieme dei valori*. Ogni chiave presente nella mappa è associata a un unico valore, ma un valore può essere associato a più chiavi.

Così come ci sono due realizzazioni dell'insieme, la libreria di Java fornisce due realizzazioni anche per le mappe: `HashMap` e `TreeMap`. Entrambe implementano l'interfaccia `Map` (si veda la Figura 3) e, come nel caso degli insiemi, dovete decidere quale usare: come regola di base, usate una mappa di tipo `HashMap`, tranne quando volete visitare le chiavi in ordine.

Dopo aver costruito un oggetto di tipo `HashMap` o `TreeMap`, dovreste memorizzare il riferimento all'oggetto mappa in una variabile di tipo `Map`:

```
Map<String, Color> favoriteColors = new HashMap<String, Color>();
```

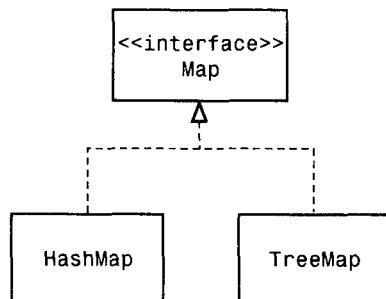
oppure

```
Map<String, Color> favoriteColors = new TreeMap<String, Color>();
```

Per aggiungere un'associazione alla mappa si usa il metodo `put`:

```
favoriteColors.put("Juliet", Color.RED);
```

**Figura 3**  
Classi e interfacce  
per le mappe  
nella libreria standard



Potete modificare il valore di un'associazione già esistente, semplicemente invocando di nuovo il metodo `put` con la stessa chiave:

```
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 rimuovere una chiave e il valore a essa associato, si usa il metodo `remove`:

```
favoriteColors.remove("Juliet");
```

**Per trovare tutte le chiavi e i valori presenti in una mappa, si itera nell'insieme delle chiavi e si cerca il valore corrispondente a ciascuna chiave.**

A volte capita di voler esaminare tutte le chiavi di una mappa, a una a una. Il metodo `keySet` restituisce un insieme contenente le chiavi, dopodiché potete chiedere un iteratore all'insieme delle chiavi e, da questo, ottenere tutte le chiavi, una dopo l'altra. Infine, per ogni chiave, si può trovare il valore associato con il metodo `get`. Quindi, le istruzioni seguenti visualizzano tutte le coppie chiave/valore presenti in una mappa `m`:

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
    Color value = m.get(key);
    System.out.println(key + " : " + value);
}
```

Quando usate una mappa di tipo `HashMap` le chiavi vengono visitate in ordine apparentemente casuale, mentre se la mappa è di tipo `TreeMap` la visita delle chiavi avviene ordinatamente. Il programma di esempio che segue mostra una mappa in azione.

### File ch15/map/MapDemo.java

```
import java.awt.Color;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

/**
 * Questo programma collauda una mappa che associa nomi a colori.
 */
public class MapDemo
{
    public static void main(String[] args)
    {
        Map<String, Color> favoriteColors = new HashMap<String, Color>();
        favoriteColors.put("Juliet", Color.BLUE);
        favoriteColors.put("Romeo", Color.GREEN);
        favoriteColors.put("Adam", Color.RED);
        favoriteColors.put("Eve", Color.BLUE);

        Set<String> keySet = favoriteColors.keySet();
        for (String key : keySet)
```

```

        {
            Color value = favoriteColors.get(key);
            System.out.println(key + " : " + value);
        }
    }
}

```

## Esecuzione del programma

```

Romeo->java.awt.Color[r=0,g=255,b=0]
Eve->java.awt.Color[r=0,g=0,b=255]
Adam->java.awt.Color[r=255,g=0,b=0]
Juliet->java.awt.Color[r=0,g=0,b=255]

```



## Auto-valutazione

5. Che differenza c'è tra un insieme e una mappa?
6. Perché le chiavi di una mappa costituiscono un insieme?



## Argomenti avanzati 15.1

### Miglioramenti alle classi contenitore in Java 7

La versione 7 di Java mette a disposizione alcune scorciatoie sintattiche per l'utilizzo di classi di tipo contenitore (*collection*).

Nei costruttori, i tipi parametrici possono essere dedotti dai tipi delle variabili: non c'è più bisogno di ripeterli nella dichiarazione delle variabili e nei costruttori. Ecco un esempio:

```

Set<String> names = new HashSet(); // costruisce un oggetto di tipo
                                   // HashSet<String>
Map<String, Integer> scores = new TreeMap(); // costruisce un
  // TreeMap<String, Integer>

```

Inoltre, con questa sintassi potete disporre di *contenitori costanti (o letterali)*, di tipo *List*, *Set* e *Map*:

```

["Tom", "Diana", "Harry"];
{ 2, 3, 5, 7, 11 };
{ "Juliet" : Color.BLUE, "Romeo" : Color.GREEN, "Eve" : Color.BLUE };

```

Questi oggetti sono immutabili, quindi non potete modificarne il contenuto: sono esemplari di classi che realizzano, rispettivamente, l'interfaccia *List*, *Set* o *Map*, ma non sapete quali classi siano.

Potete usare un contenitore costante come argomento di un metodo, come in questo esempio:

```
names.addAll(["Tom", "Diana", "Harry"]);
```

Se volete memorizzare un contenitore costante in una variabile, nella sua dichiarazione dovete usare il tipo dell'interfaccia:

```
List<String> friends = ["Tom", "Diana", "Harry"];
```

In alternativa, potete creare un contenitore usando un contenitore costante come parametro di costruzione:

```
ArrayList<String> friends = new ArrayList<>(["Tom", "Diana", "Harry"]);
```

Questo funziona perché, in Java, tutte le classi di tipo contenitore o mappa hanno costruttori che copiano gli elementi da un altro contenitore o mappa.

Infine, invece di usare i metodi `get`, `set` o `put`, potete utilizzare l'operatore `[]`:

```
String name = names[0];
names[0] = "Fred";
scores["Fred"] = 13;
int score = scores["Fred"];
```



## Consigli pratici 15.1

### Scegliere un contenitore

Immaginate di dover memorizzare oggetti in un contenitore; a questo punto, avete visto un certo numero di diverse strutture per i dati e questi “Consigli pratici” riassumono come si possa scegliere il contenitore più appropriato per un’applicazione.

#### Fase 1 Determinate le modalità di accesso agli elementi

Gli elementi vengono memorizzati in un contenitore per poterli, in seguito, recuperare. Come volete accedere ai singoli elementi? Ci sono diverse possibilità.

- Si accede ai valori mediante una posizione, rappresentata da un numero intero. In tal caso usate un contenitore di tipo `ArrayList`, passate alla Fase 2 e, quindi, avrete finito.
- Si accede ai valori mediante una chiave che non fa parte dell’oggetto: usate una mappa.
- Non importa. Si accede sempre ai valori “in blocco”, visitandoli tutti e facendo qualcosa con ciascuno di essi.

#### 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 recuperate gli elementi dal contenitore o le chiavi dalla mappa, vi interessa l'ordine con il quale vengono estratti? Ci sono diverse possibilità.

- Gli elementi o le chiavi devono essere in ordine: usate un oggetto di tipo `TreeSet` o `TreeMap` e passate alla Fase 6.
- Gli elementi devono essere nello stesso ordine in cui sono stati inseriti: la scelta si restringe ai soli esemplari di `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, usatene una di tipo `HashMap` e passate alla Fase 5.

**Fase 4** Per un contenitore che non sia una mappa, determinate quali operazioni devono essere veloci

Ci sono diverse possibilità.

- La ricerca di elementi deve essere veloce: usate un contenitore di tipo `HashSet` e passate alla Fase 5.
- L'aggiunta e la rimozione di elementi nella posizione iniziale o in posizioni intermedie devono essere veloci: usate un contenitore di tipo `LinkedList`.
- Non importa. Effettuate inserimenti soltanto nella posizione terminale e usate così pochi elementi che la velocità non vi interessa: usate un contenitore di tipo `ArrayList`.

**Fase 5** Per insiemi e mappe, scegliete se occorre realizzare i metodi `equals` e `hashCode`

Se i vostri elementi (o chiavi) appartengono a una classe definita da qualcun altro, controllate se realizza i metodi `hashCode` e `equals`: se è così, siete a posto. Questo accade per la maggior parte delle classi presenti nella libreria standard di Java, come `String`, `Integer`, `Rectangle` e così via.

In caso contrario, decidete se potete confrontare gli elementi per identità: nel vostro programma, gli elementi sono tutti diversi? Detto in altro modo: è vero che non può accadere che due elementi distinti abbiano le variabili di esemplare identiche? In tal caso non avete bisogno di far nulla: i metodi `hashCode` e `equals` della classe `Object` sono adeguati.

Se dovete, invece, realizzare vostri metodi `hashCode` e `equals`, consultate il Paragrafo 15.4.

**Fase 6** Se usate un albero, decidete se serve un comparatore

Osservate la classe degli elementi o delle chiavi. La classe realizza l'interfaccia `Comparable`? In tal caso, se l'ordinamento generato dal metodo `compareTo` è quello che volete, allora non avete bisogno di fare nient'altro. Questo accade per molte classi presenti nella libreria standard, in particolare per le classi `String` e `Integer`.

In caso contrario, la classe dei vostri elementi deve realizzare l'interfaccia `Comparable` oppure dovete progettare una classe che realizzzi l'interfaccia `Comparator`, come visto nel Paragrafo 13.8.



## Esempi completi 15.1

### Determinare la frequenza di parole in un testo

In questo esempio completo leggiamo un file di testo e visualizziamo un elenco contenente, in ordine alfabetico, tutte le parole presenti nel file, seguite da un conteggio che indica quanto spesso ciascuna parola ricorre nel file.

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

Nel nostro caso i valori sono le frequenze delle parole e abbiamo un valore di frequenza per ciascuna parola, per cui vogliamo usare una mappa, che consenta di trovare la frequenza associata a una parola.

#### Fase 2 Determinate il tipo degli elementi o i tipi di chiavi e valori

Le parole sono di tipo `String` e le frequenze sono di tipo `Integer` (non possiamo usare `int` come tipo parametrico perché è un tipo primitivo), per cui 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 una mappa di tipo `TreeMap`.

#### Fase 4 Per un contenitore che non sia una mappa, determinate quali operazioni devono essere veloci

Saltiamo questa fase, perché usiamo una mappa, non un contenitore.

#### Fase 5 Per insiemi e mappe, scegliete se occorre realizzare i metodi `equals` e `hashCode`

Saltiamo questa fase, perché usiamo una mappa di tipo `TreeMap`.

#### Fase 6 Se usate un albero, decidete se serve un comparatore

Il tipo delle chiavi della nostra mappa è `String`, che realizza l'interfaccia `Comparable`, per cui non dobbiamo fare nient'altro.

Abbiamo quindi scelto la nostra struttura di memorizzazione dei dati. Ora, 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 (es: segni di punteggiatura).

  Se la parola è già presente nella mappa delle frequenze

    Incrementa la frequenza associata.

  Altrimenti

    Imposta la frequenza al valore 1.

Ecco il codice del programma:

### File ch15/wordfreq/WordFrequency.java

```
import java.util.Map;
import java.util.Scanner;
import java.util.TreeMap;
import java.io.File;
import java.io.FileNotFoundException;

public class WordFrequency
{
    /**
     * Elimina da una stringa i caratteri che non sono lettere.
     * @param s una stringa
     * @return una stringa contenente tutte le lettere presenti in s
     */
    public static String clean(String s)
    {
        String r = "";
        for (int i = 0; i < s.length(); i++)
        {
            char c = s.charAt(i);
            if (Character.isLetter(c))
            {
                r = r + c;
            }
        }
        return r.toLowerCase();
    }

    public static void main(String[] args)
        throws FileNotFoundException
    {
        Map<String, Integer> frequencies = new TreeMap<String, Integer>();
        Scanner in = new Scanner(new File("alice30.txt"));
        while (in.hasNext())
        {
            String word = clean(in.next());
            Integer count = frequencies.get(word);
            if (count == null) { count = 1; }
            else { count = count + 1; }
            frequencies.put(word, count);
        }

        for (String key : frequencies.keySet())
        {
```

```
        System.out.printf("%-20s%10d\n", key, frequencies.get(key));
    }
}
```

### 15.3 Tabelle hash

In questo paragrafo vedrete come si possa usare la tecnica dell'*hashing* per cercare velocemente elementi in una struttura per dati, senza fare una ricerca lineare tra tutti gli elementi stessi. Il procedimento di *hashing* dà origine a una *tabella hash*, che può essere usata per realizzare insiemi e mappe.

Una funzione di hash calcola un valore intero a partire da un oggetto.

Una funzione di hash è una funzione che, a partire da un oggetto, calcola un valore intero, il codice di hash, facendo in modo che oggetti diversi abbiano con buona probabilità codici di hash diversi. La classe Object ha un metodo hashCode che le altre classi devono sovrascrivere. L'invocazione

```
int h = x.hashCode();
```

calcola il codice di hash dell'oggetto x.

**Tabella 1**  
Codici di hash  
di alcune stringhe

| Stringa     | Codice di hash |
|-------------|----------------|
| "Adam"      | 2035631        |
| "Eve"       | 70068          |
| "Harry"     | 69496448       |
| "Jim"       | 74478          |
| "Joe"       | 74656          |
| "Juliet"    | -2065036585    |
| "Katherine" | 2079199209     |
| "Sue"       | 83491          |

Una buona funzione di hash minimizza le *collisioni*, che avvengono quando a oggetti diversi vengono associati codici di hash identici.

La Tabella 1 mostra alcuni esempi di stringhe e dei loro codici di hash. Vedrete nel Paragrafo 15.4 come sono stati ottenuti questi valori.

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 P15.6).

Il Paragrafo 15.4 spiega anche come sovrascrivere il metodo `hashCode` per altre classi.

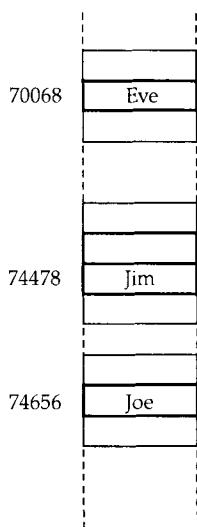
All'interno di una tabella hash, un codice di hash viene usato come indice di un array.

All'interno di una tabella hash, un codice di hash viene usato come indice di un array. Nella realizzazione più semplice di una tabella hash, potreste costruire un array e inserire ciascun oggetto nella posizione indicata dal suo codice di hash, come si può vedere nella Figura 4.

Se non ci sono collisioni, è molto semplice scoprire se un oggetto sia già presente nell'insieme oppure no: calcolate il suo codice di hash e verificate se la posizione dell'array indicata da quel codice è già occupata. Non serve una ricerca all'interno dell'array!

Ovviamente, però, non è possibile creare un array che sia tanto grande da contenere le posizioni indicate da tutti i numeri interi possibili. Quindi, occorre scegliere un array

**Figura 4**  
Una realizzazione semplificata di tabella hash

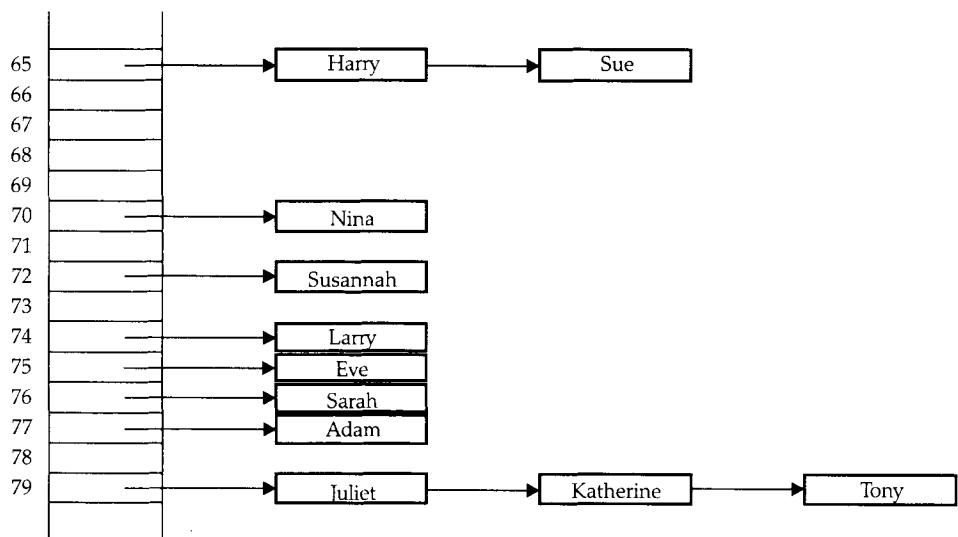


con una qualche ragionevole dimensione e costringere il codice di hash a ricadere in tale intervallo:

```
int h = x.hashCode();
if (h < 0) h = -h;
position = h % buckets.length;
```

Dopo aver ricondotto il codice di hash all'interno della relativamente piccola dimensione dell'array mediante l'operazione modulo, diventa ancora più probabile che più oggetti collidano. Per gestire le collisioni, memorizzeremo tutti gli oggetti collidenti in una lista concatenata, chiamata *bucket*, contenente tutti gli elementi che hanno lo stesso codice di hash (si veda la Figura 5).

**Figura 5**  
Una tabella hash con *bucket*, ciascuno dei quali memorizza elementi aventi lo stesso codice di hash



Una tabella hash può essere realizzata come un array di *bucket*, sequenze di nodi che contengono elementi aventi lo stesso codice di hash.

Se non ci sono collisioni o se ce ne sono poche, aggiungere, eliminare e cercare elementi nella tabella hash richiede un tempo costante,  $O(1)$ .

Ecco, infine, l'algoritmo per cercare un oggetto  $x$  in una tabella hash.

1. Si calcola il codice di hash di  $x$  e lo si riduce modulo la dimensione della tabella, generando un indice  $h$  all'interno della tabella hash.
2. Si itera attraverso gli elementi del bucket che si trova in posizione  $h$ : per ogni elemento del bucket, si verifica se è uguale a  $x$ .
3. Se si trova una corrispondenza fra gli elementi di tale bucket, allora  $x$  appartiene all'insieme, altrimenti no.

L'inserimento di un elemento è una semplice estensione dell'algoritmo utilizzato per cercare un oggetto. Dapprima si calcola il codice di hash, per trovare il bucket in cui l'elemento dovrebbe essere inserito, poi si cerca l'elemento in quel bucket: se è già presente, non si fa nulla, altrimenti lo si inserisce.

Eliminare un elemento è altrettanto semplice. Dapprima si calcola il codice di hash per trovare il bucket in cui dovrebbe trovarsi l'elemento, poi si cerca l'elemento in quel bucket: se è presente, lo si rimuove, altrimenti non si fa nulla.

Nel caso migliore, quando non vi sono collisioni, tutti i bucket sono vuoti o hanno un solo elemento, per cui le operazioni di inserimento, rimozione e ricerca richiedono un tempo costante,  $O(1)$ .

Più in generale, affinché questo algoritmo sia efficiente, le dimensioni dei bucket devono essere piccole. Nel caso peggiore, quando tutti gli elementi vanno a finire nello stesso bucket, una tabella hash degenera in una lista concatenata!

Per ridurre la probabilità di collisione, occorre creare una tabella un po' più grande del numero di elementi che si prevede di inserire: una capacità eccedente del 30% è un valore tipico. Secondo alcuni ricercatori, al fine di minimizzare il numero di collisioni, la dimensione di una tabella hash dovrebbe essere scelta in modo da essere un numero primo.

Al termine di questo paragrafo viene proposto il codice per una semplice realizzazione con tabella hash di un insieme, sfruttando la classe `AbstractSet`, che realizza già la maggior parte dei metodi dell'interfaccia `Set`.

In questa realizzazione occorre specificare la dimensione della tabella hash. Nella libreria standard non c'è, invece, bisogno di fornire una dimensione per la tabella: se la tabella hash si riempie troppo, viene creata una nuova tabella di dimensione doppia e tutti gli elementi vengono inseriti nella nuova tabella.

### File ch15/hashtable/HashSet.java

```
import java.util.AbstractSet;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * Un esemplare di HashSet memorizza una raccolta
 * non ordinata di oggetti usando una tabella hash.
 */
public class HashSet extends AbstractSet
{
    private Node[] buckets;
```

```
private int size;

/**
 * Costruisce una tabella hash.
 * @param bucketsLength la lunghezza dell'array di bucket
 */
public HashSet(int bucketsLength)
{
    buckets = new Node[bucketsLength];
    size = 0;
}

/**
 * Verifica l'appartenenza all'insieme.
 * @param x un oggetto
 * @return true se x è un elemento dell'insieme
 */
public boolean contains(Object x)
{
    int h = x.hashCode();
    if (h < 0) h = -h;
    h = h % buckets.length;

    Node current = buckets[h];
    while (current != null)
    {
        if (current.data.equals(x)) return true;
        current = current.next;
    }
    return false;
}

/**
 * Aggiunge un elemento all'insieme.
 * @param x un oggetto
 * @return true se x è un oggetto nuovo, false se x era già
 *         presente nell'insieme
 */
public boolean add(Object x)
{
    int h = x.hashCode();
    if (h < 0) h = -h;
    h = h % buckets.length;

    Node current = buckets[h];
    while (current != null)
    {
        if (current.data.equals(x))
            return false; // già presente nell'insieme
        current = current.next;
    }
    Node newNode = new Node();
    newNode.data = x;
    newNode.next = buckets[h];
    buckets[h] = newNode;
    size++;
}
```

```
        return true;
    }

    /**
     * Elimina un oggetto dall'insieme.
     * @param x un oggetto
     * @return true se x viene eliminato dall'insieme,
     *         false se x non è presente nell'insieme
    */
    public boolean remove(Object x)
    {
        int h = x.hashCode();
        if (h < 0) h = -h;
        h = h % buckets.length;

        Node current = buckets[h];
        Node previous = null;
        while (current != null)
        {
            if (current.data.equals(x))
            {
                if (previous == null)
                    buckets[h] = current.next;
                else previous.next = current.next;
                size--;
                return true;
            }
            previous = current;
            current = current.next;
        }
        return false;
    }

    /**
     * Restituisce un iteratore che visita gli elementi dell'insieme.
     * @return un iteratore per insiemi realizzati con tabella hash
    */
    public Iterator iterator()
    {
        return new HashSetIterator();
    }

    /**
     * Restituisce il numero di elementi nell'insieme.
     * @return il numero di elementi
    */
    public int size()
    {
        return size;
    }

    class Node
    {
        public Object data;
        public Node next;
    }
}
```

```
class HashSetIterator implements Iterator
{
    private int bucket;
    private Node current;
    private int previousBucket;
    private Node previous;

    /**
     * Costruisce un iteratore per insiemi realizzati con tabella
     * hash che punta al primo elemento dell'insieme.
     */
    public HashSetIterator()
    {
        current = null;
        bucket = -1;
        previous = null;
        previousBucket = -1;
    }

    public boolean hasNext()
    {
        if (current != null && current.next != null)
            return true;
        for (int b = bucket + 1; b < buckets.length; b++)
            if (buckets[b] != null) return true;
        return false;
    }

    public Object next()
    {
        previous = current;
        previousBucket = bucket;
        if (current == null || current.next == null)
        {
            // avanza al prossimo bucket
            bucket++;

            while (bucket < buckets.length
                  && buckets[bucket] == null)
                bucket++;
            if (bucket < buckets.length)
                current = buckets[bucket];
            else
                throw new NoSuchElementException();
        }
        else // avanza al prossimo elemento nel bucket
            current = current.next;
        return current.data;
    }

    public void remove()
    {
        if (previous != null && previous.next == current)
            previous.next = current.next;
        else if (previousBucket < bucket)
            buckets[bucket] = current.next;
    }
}
```

```
        else
            throw new IllegalStateException();
        current = previous;
        bucket = previousBucket;
    }
}
```

## File ch15/hashtable/HashSetDemo.java

```
import java.util.Iterator;
import java.util.Set;

/**
 Questo programma collauda la classe che realizza
 un insieme con una tabella hash.
 */
public class HashSetDemo
{
    public static void main(String[] args)
    {
        Set names = new HashSet(101); // 101 è un numero primo

        names.add("Harry");
        names.add("Sue");
        names.add("Nina");
        names.add("Susannah");
        names.add("Larry");
        names.add("Eve");
        names.add("Sarah");
        names.add("Adam");
        names.add("Tony");
        names.add("Katherine");
        names.add("Juliet");
        names.add("Romeo");
        names.remove("Romeo");
        names.remove("George");

        Iterator iter = names.iterator();
        while (iter.hasNext())
            System.out.println(iter.next());
    }
}
```

## Esecuzione del programma

Harry  
Sue  
Nina  
Susannah  
Larry  
Eve  
Sarah  
Adam

Juliet  
Katherine  
Tony



## Auto-valutazione

7. Se una funzione di hash restituisse 0 per qualsiasi valore, la classe `HashSet` funzionerebbe comunque in modo corretto?
8. Cosa fa il metodo `hasNext` della classe `HashSetIterator` quando viene raggiunta la fine di un bucket?

## 15.4 Calcolare codici hash

Una funzione di hash calcola un codice di hash intero a partire da un oggetto, in modo che sia probabile che oggetti diversi abbiano un codice diverso. Per prima cosa vediamo come si possa calcolare un codice di hash per una stringa. Chiaramente, bisogna combinare i valori dei caratteri della stringa per ottenere un numero intero; si potrebbe, ad esempio, sommare i valori dei caratteri:

```
int h = 0;
for (int i = 0; i < s.length(); i++)
    h = h + s.charAt(i);
```

Questa, però, non sarebbe una buona idea, perché non mescola sufficientemente i valori dei caratteri: stringhe che sono permutazioni di un'altra (come "eat" e "tea") avrebbero tutte lo stesso codice di hash.

Ecco invece 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, il codice di hash di "eat" è

$31 * (31 * 'e' + 'a') + 't' = 100184$

Il codice di hash di "tea" è abbastanza diverso

$31 * (31 * 't' + 'e') + 'a' = 114704$

Per realizzare il metodo  
`hashCode`, combinate i codici  
di hash delle variabili di esemplare.

Per le vostre classi dovreste definire un codice di hash che combini in modo simile i codici di hash delle variabili di esemplare. Ad esempio, definiamo il metodo `hashCode` per la classe `Coin`, che ha due variabili di esemplare: il nome della moneta e il suo valore. Per prima cosa calcoliamo il loro codice 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 oggetto di tipo `Double` che lo contenga, poi calcolate il suo codice di hash.

```

class Coin
{
    public int hashCode()
    {
        int h1 = name.hashCode();
        int h2 = new Double(value).hashCode();
        ...
    }
}

```

Combinate quindi i due codici di hash.

```

final int HASH_MULTIPLIER = 29;
int h = HASH_MULTIPLIER * h1 + h2;
return h;

```

Usate un numero primo come moltiplicatore di hash, perché mescola meglio i valori.

Se avete più di due variabili di esemplare, combinate i loro codici di hash in questo modo:

```

int h = HASH_MULTIPLIER * h1 + h2;
h = HASH_MULTIPLIER * h + h3;
h = HASH_MULTIPLIER * h + h4;
...
return h;

```

Se una delle variabili di esemplare è un numero intero, usatelo direttamente come suo codice di hash.

**Il metodo hashCode deve essere compatibile con il metodo equals.**

Quando aggiungete a una tabella di hash oggetti di una vostra classe, dovete controllarci attentamente che, nella classe, il metodo `hashCode` sia *compatibile* con il metodo `equals`. Due oggetti uguali devono avere lo stesso codice di hash:

- Se `x.equals(y)`, allora `x.hashCode() == y.hashCode()`

Dopo tutto, se `x` e `y` sono uguali, non volete inserirli entrambi in un insieme: gli insiemi non conservano duplicati. Ma, se i loro codici di hash sono diversi, `x` e `y` possono andare a finire in bucket diversi e il metodo `add` non si accorgerà mai che, in realtà, sono duplicati.

Ovviamente, in generale il contrario della condizione di compatibilità non è vero: è possibile che due oggetti abbiano lo stesso codice di hash senza essere uguali.

Per la classe `Coin` la condizione di compatibilità è valida: diciamo che due monete sono uguali se i loro nomi e i loro valori sono uguali, ma, in tal caso, anche i loro codici di hash saranno uguali, perché il codice di hash viene calcolato usando i codici di hash delle variabili `name` e `value`, che contengono il nome e il valore.

Se la vostra classe sovrascrive il metodo `equals` ma non il metodo `hashCode` ci saranno dei problemi. Supponiamo di esserci dimenticati di definire il metodo `hashCode` per la classe `Coin`: di conseguenza, essa eredita il metodo per il calcolo del codice di hash 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.

```
Coin coin1 = new Coin(0.25, "quarter");
Coin coin2 = new Coin(0.25, "quarter");
```

Ora, `coin1.hashCode()` viene calcolato a partire dall'indirizzo di memoria di `coin1`, mentre `coin2.hashCode()` viene calcolato a partire dall'indirizzo di memoria di `coin2`. Nonostante `coin1.equals(coin2)` sia `true`, i loro codici di hash sono (molto probabilmente) diversi.

Se una classe non sovrascrive né `equals` né `hashCode`, due suoi esemplari sono uguali soltanto quando sono lo stesso oggetto.

Tuttavia, se non definite né `equals` né `hashCode`, non ci sono problemi. Il metodo `equals` della classe `Object` considera due oggetti uguali solo se sono, in realtà, lo stesso oggetto, quindi hanno lo stesso indirizzo in memoria e, 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. Questa non è necessariamente una cattiva definizione di uguaglianza: se volete raccogliere un insieme di monete in un borsellino, non vorrete che monete dello stesso valore siano considerate uguali.

Ogni volta che usate un insieme realizzato con tabella hash dovete essere certi che esista una funzione di hash adatta per il tipo di oggetti che volete inserire nell'insieme. Verificate il metodo `equals` della vostra classe, che vi dice quando due oggetti vengono considerati uguali. Se `equals` non è stato sovrascritto nella classe, due suoi esemplari vengono considerati uguali soltanto se sono lo stesso oggetto: in tal caso, non definite nemmeno il metodo `hashCode`. Se, invece, il metodo `equals` è stato sovrascritto, osservate la sua realizzazione. Tipicamente, due oggetti vengono considerati uguali se tutte o alcune variabili di esemplare sono uguali. A volte, non tutte le variabili vengono usate nel confronto: ad esempio, due oggetti di tipo `Student` possono considerarsi uguali se le loro variabili `studentID` (un “numero di matricola” univoco) sono uguali. Definite il metodo `hashCode` in modo che combini i codici di hash delle stesse variabili che vengono confrontate nel metodo `equals`.

Quando usate una mappa di tipo `HashMap`, vengono calcolati i codici di hash soltanto per le chiavi, che devono avere metodi `equals` e `hashCode` compatibili. I valori non vengono confrontati, né vengono calcolati i loro codici di hash. Il motivo è semplice: la mappa ha bisogno di aggiungere, rimuovere e cercare velocemente soltanto le chiavi.

### File ch15/hashcode/Coin.java

```
/*
 * Una moneta con un valore.
 */
public class Coin
{
    private double value;
    private String name;

    /**
     * Costruisce una moneta.
     * @param aValue il valore della moneta
     * @param aName il nome della moneta
     */
    public Coin(double aValue, String aName)
    {
        value = aValue;
```

```

        name = aName;
    }

    /**
     * Restituisce il valore della moneta.
     * @return il valore della moneta
    */
    public double getValue()
    {
        return value;
    }

    /**
     * Restituisce il nome della moneta.
     * @return il nome della moneta
    */
    public String getName()
    {
        return name;
    }

    public boolean equals(Object otherObject)
    {
        if (otherObject == null) return false;
        if (getClass() != otherObject.getClass()) return false;
        Coin other = (Coin) otherObject;
        return (value == other.value && name.equals(other.name));
    }

    public int hashCode()
    {
        int h1 = name.hashCode();
        int h2 = new Double(value).hashCode();
        final int HASH_MULTIPLIER = 29;
        int h = HASH_MULTIPLIER * h1 + h2;
        return h;
    }

    public String toString()
    {
        return "Coin[value=" + value + ",name=" + name + "]";
    }
}

```

### File ch15/hashcode/CoinHashCodePrinter.java

```

import java.util.HashSet;
import java.util.Set;

/**
 * Un programma per visualizzare i codici di hash di monete.
 */
public class CoinHashCodePrinter
{
    public static void main(String[] args)
    {

```

```

Coin coin1 = new Coin(0.25, "quarter");
Coin coin2 = new Coin(0.25, "quarter");
Coin coin3 = new Coin(0.05, "nickel");

System.out.println("hash code of coin1=" + coin1.hashCode());
System.out.println("hash code of coin2=" + coin2.hashCode());
System.out.println("hash code of coin3=" + coin3.hashCode());

Set<Coin> coins = new HashSet<Coin>();
coins.add(coin1);
coins.add(coin2);
coins.add(coin3);

for (Coin c : coins)
    System.out.println(c);
}
}

```

### Esempio di esecuzione del programma

```

hash code of coin1=-1513525892
hash code of coin2=-1513525892
hash code of coin3=-768365211
Coin[value=0.25,name=quarter]
Coin[value=0.05,name.nickel]

```



### Auto-valutazione

9. Qual è il codice di hash della stringa "to"?
10. Qual è il codice di hash dell'oggetto new Integer(13)?



### Errori comuni 15.1

#### Dimenticarsi di sovrascrivere hashCode

Quando inserite elementi in una tabella hash, accertatevi che il metodo `hashCode` sia stato sovrascritto. Eccezione: non c'è bisogno di sovrascrivere il metodo `hashCode` se anche `equals` non è sovrascritto, però, in tal caso, due oggetti distinti della vostra classe vengono considerati diversi anche se hanno lo stesso contenuto.

Se dimenticate di realizzare il metodo `hashCode`, viene ereditato il metodo `hashCode` della classe `Object`, che calcola un codice di hash a partire dall'indirizzo di memoria dell'oggetto. Ad esempio, supponiamo di *non* aver sovrascritto il metodo `hashCode` nella classe `Coin`. In questo caso, è probabile che il codice seguente fallisca:

```

Set<Coin> coins = new HashSet<Coin>();
coins.add(new Coin(0.25, "quarter"));
// il confronto che segue probabilmente fallirà
// se hashCode non è stato sovrascritto in Coin
if (coins.contains(new Coin(0.25, "quarter")))
    System.out.println("The set contains a quarter.");

```

Quando i due oggetti di tipo `Coin` vengono costruiti, sono loro assegnati indirizzi in memoria diversi, per cui il metodo `hashCode` della classe `Object` probabilmente calcolerà per loro codici di hash diversi (come sempre succede con i codici di hash, esiste una remota possibilità che i codici di hash collidano). Di conseguenza, il metodo `contains` ispezionerà il bucket sbagliato e non troverà mai la moneta corrispondente.

Il rimedio consiste nel sovrascrivere il metodo `hashCode` nella classe `Coin`.

## 15.5 Alberi di ricerca binari

Ogni particolare realizzazione di un insieme può sistemare i propri elementi come preferisce, per poterli ritrovare velocemente. Supponiamo che, ad esempio, li mantenga *ordinati*: di conseguenza, può usare la *ricerca binaria* per trovarli velocemente. La ricerca binaria richiede  $O(\log(n))$  passi, dove  $n$  è la dimensione dell'insieme. Ad esempio, la ricerca binaria in un array di 1000 elementi è in grado di trovare un elemento in circa 10 passi, dividendo a metà la dimensione dell'intervallo di ricerca a ogni passo.

Se, però, usiamo un array per memorizzare gli elementi di un insieme secondo un determinato criterio di ordinamento, l'inserimento e la rimozione sono operazioni  $O(n)$ . In questo paragrafo vedrete come strutture di dati *a forma di albero* possano conservare gli elementi di un insieme in ordine, consentendo inserimenti e rimozioni più efficienti.

Una lista concatenata è una struttura con una sola dimensione: ciascun nodo ha un collegamento al nodo che lo segue e potete immaginare che tutti i nodi siano disposti su una sola riga. Per contro, un *albero* è costituito da nodi che hanno più riferimenti ad altri nodi, chiamati *figli*. Poiché i figli possono, a loro volta, avere figli, la struttura dei dati assomiglia a un albero, che viene per tradizione disegnato con la “radice” in alto, come un albero genealogico o un organigramma (osservate la Figura 6). Continuando con l'allegoria, il nodo che si trova in cima all'albero è detto *nodo radice* (“root node”), mentre i nodi privi di figli si dicono *nodi foglie* (“leaf node”). In un *albero binario*, ogni nodo ha al massimo due figli (chiamati *figlio sinistro* e *figlio destro*), da cui il nome *binario*.

Infine, un *albero di ricerca binario* (“binary search tree”) viene costruito in modo che abbia la seguente importante proprietà:

- I valori dei dati di *tutti* i discendenti che si trovano alla sinistra di *qualsiasi* nodo sono inferiori al valore del dato memorizzato in quel nodo, mentre *tutti* i discendenti che si trovano alla destra contengono valori maggiori.

Un albero binario è composto di nodi, ciascuno dei quali ha al massimo due nodi figli.

Tutti i nodi di un albero di ricerca binario soddisfano questa proprietà: i discendenti di sinistra di un nodo contengono dati di valore inferiore al dato contenuto nel nodo, mentre i discendenti di destra contengono dati di valore maggiore.

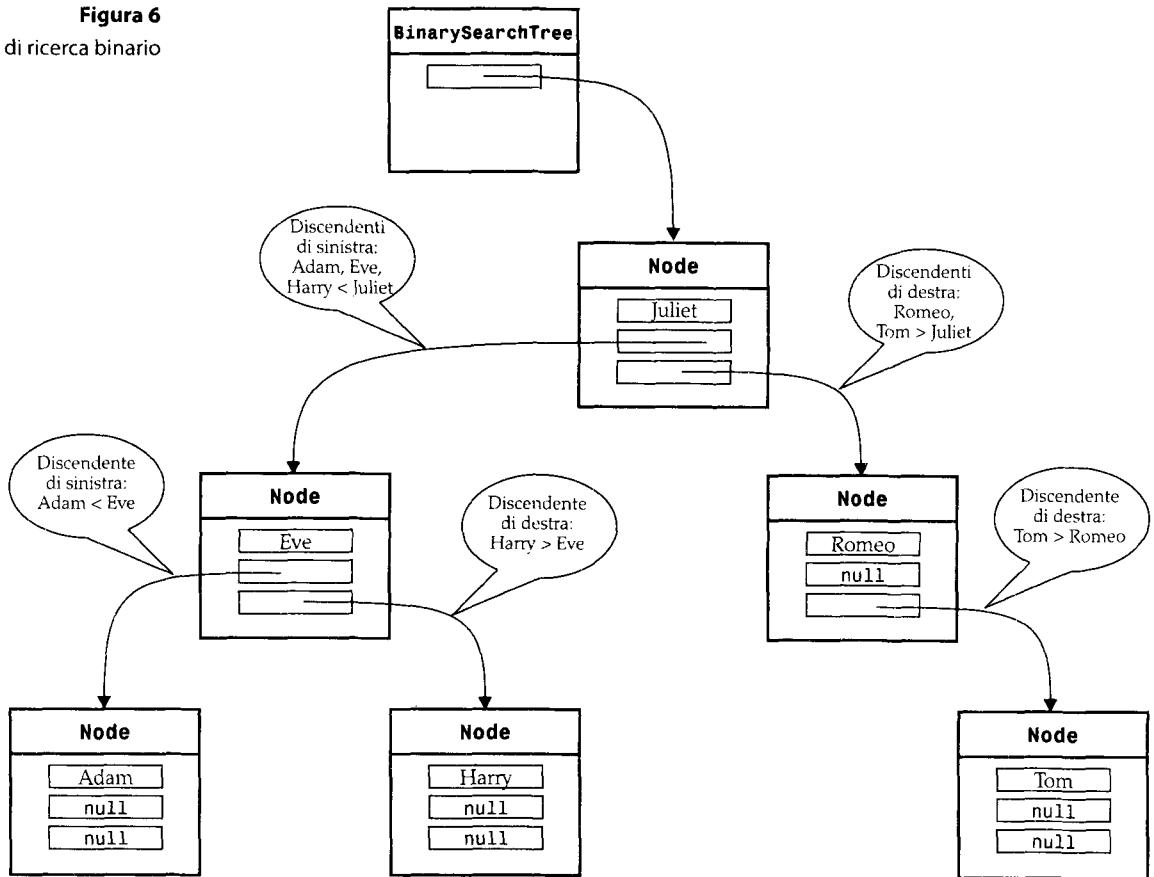
L'albero della Figura 6 ha questa proprietà, che va verificata per ciascun nodo. Considerate il nodo “Juliet”: tutti i suoi discendenti di sinistra hanno dati che precedono “Juliet” e tutti i suoi discendenti di destra hanno dati che seguono “Juliet”. Passate a “Eve”: c'è un unico discendente di sinistra, contenente “Adam”, che viene prima di “Eve”, e un solo discendente di destra, contenente “Harry”, che viene dopo “Eve”. Controllate nello stesso modo i nodi restanti.

La Figura 7 mostra un albero binario che non è un albero di ricerca binario. Osservate attentamente: il nodo radice supera la verifica, ma i suoi due figli no.

Proviamo a realizzare le classi necessarie per rappresentare questo albero. Così come avevate bisogno di classi per le liste e per i loro nodi, vi occorre una classe per l'albero, che

**Figura 6**

Un albero di ricerca binario



contenga un riferimento al *nodo radice* (“root”), e una classe separata per i nodi. Ciascun nodo contiene due riferimenti (al nodo figlio di sinistra e a quello di destra) e una variabile di esemplare *data*. Ai margini dell’albero, uno o due dei riferimenti ai figli hanno il valore *null*. La variabile *data* è di tipo *Comparable*, non *Object*, perché in un albero di ricerca binario dovete essere in grado di confrontare i valori per poterli collocare nelle posizioni corrette.

```

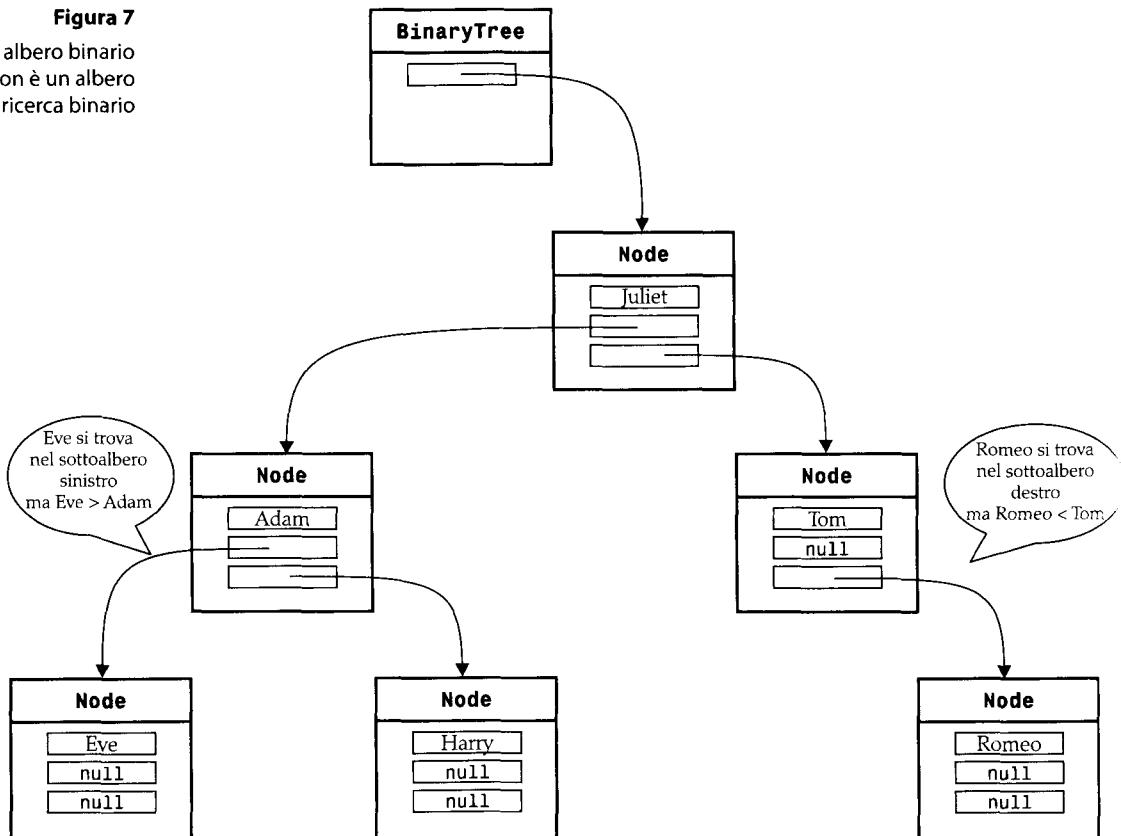
public class BinarySearchTree
{
    private Node root;

    public BinarySearchTree() { . . . }
    public void add(Comparable obj) { . . . }
    ...
    class Node
    {
        public Comparable data;
        public Node left;
        public Node right;
    }
}

```

**Figura 7**

Un albero binario  
che non è un albero  
di ricerca binario



```

public void addNode(Node newNode) { . . . }
{
    ...
}
    
```

Per inserire dati nell'albero, utilizzate il seguente algoritmo, partendo dalla radice:

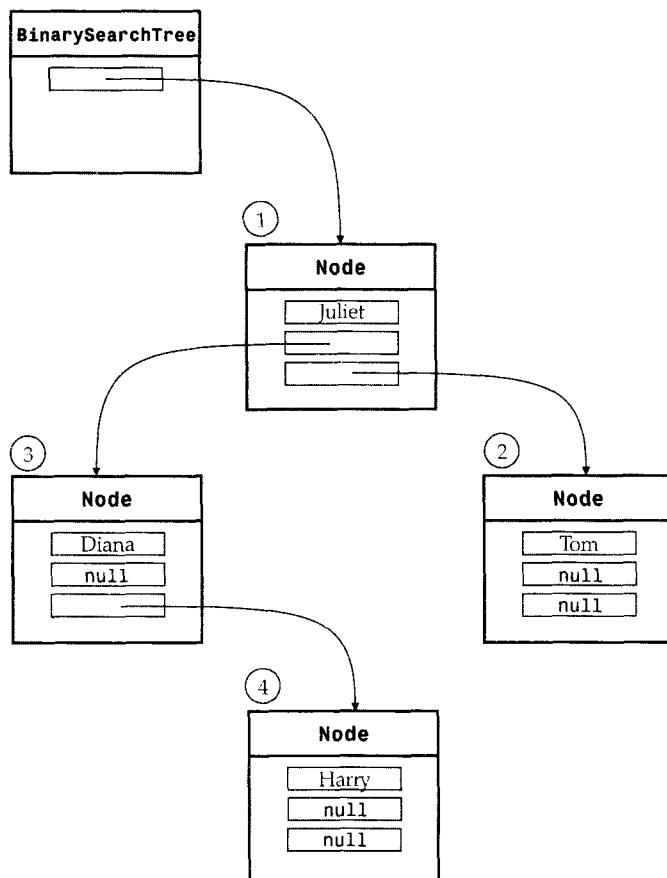
- Se trovate un riferimento a un nodo che sia diverso da `null`, esamineate la variabile `data` di quel nodo. Se il valore è maggiore di quello che volete inserire, continuate il processo nel sottoalbero sinistro, altrimenti continuate il processo nel sottoalbero destro.
- Se trovate un riferimento `null`, sostituitelo con il nuovo nodo.

Ad esempio, considerate l'albero della Figura 8, che è il risultato dell'esecuzione dei seguenti enunciati:

```

BinarySearchTree tree = new BinarySearchTree();
tree.addNode("Juliet"); // 1
tree.addNode("Tom"); // 2
tree.addNode("Diana"); // 3
tree.addNode("Harry"); // 4
    
```

**Figura 8**  
Un albero di ricerca binario  
dopo quattro inserimenti



Vogliamo inserirvi un nuovo elemento, "Romeo".

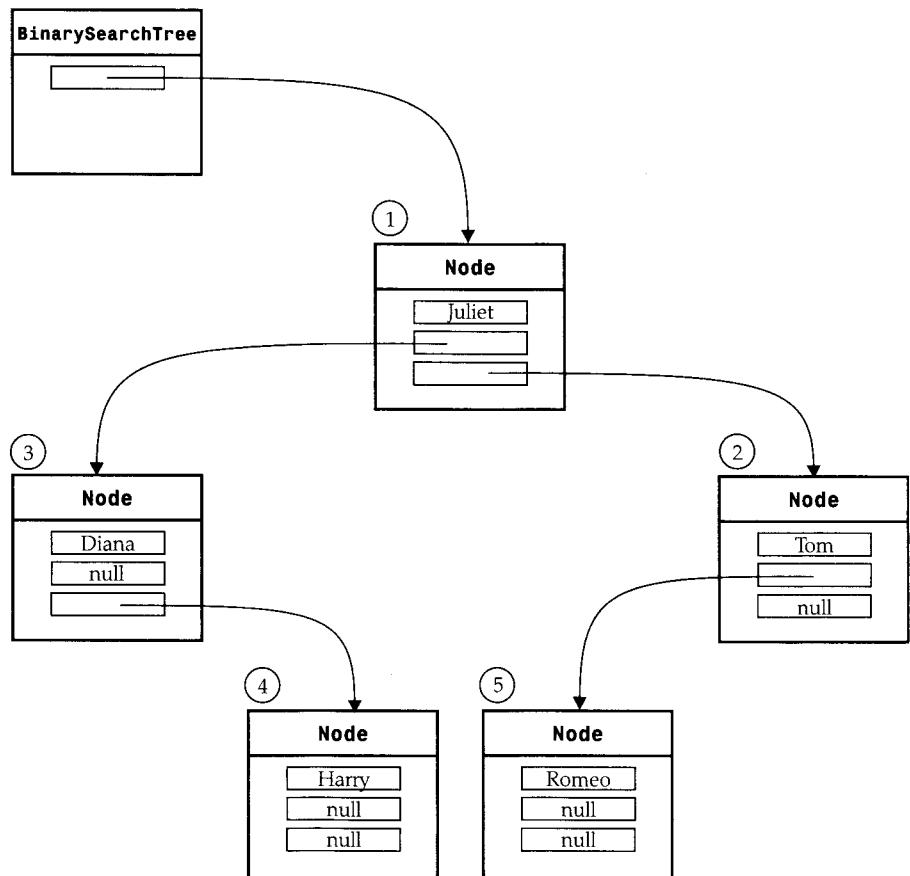
```
tree.add("Romeo"); // 5
```

Cominciate con la radice, "Juliet": "Romeo" viene dopo "Juliet", quindi spostatevi nel sottoalbero destro, dove trovate il nodo "Tom". "Romeo" viene prima di "Tom", quindi spostatevi nel suo sottoalbero sinistro. Però un sottoalbero sinistro non c'è: di conseguenza, inserite un nuovo nodo, "Romeo", come figlio sinistro di "Tom" (osservate la Figura 9).

Dovreste, a questo punto, essere convinti che l'albero che ne risulta è ancora un albero di ricerca binario. Quando viene inserito "Romeo", deve diventare un discendente destro di "Juliet": questo è ciò che la condizione dell'albero di ricerca binario richiede per il nodo radice "Juliet". Al nodo radice non interessa in che punto del proprio sottoalbero destro vada a finire il nuovo nodo. Dal punto di vista di "Tom", il figlio destro di "Juliet", l'unica cosa che interessa è che il nuovo nodo, "Romeo", finisca da qualche parte alla sua sinistra. Non c'è niente alla sua sinistra, quindi "Romeo" diventa il suo nuovo figlio sinistro e l'albero che ne risulta è ancora un albero di ricerca binario.

**Figura 9**

Un albero di ricerca binario  
dopo cinque inserimenti



Ecco il codice per il metodo add della classe `BinarySearchTree`:

```

public void add(Comparable obj)
{
    Node newNode = new Node();
    newNode.data = obj;
    newNode.left = null;
    newNode.right = null;
    if (root == null) root = newNode;
    else root.addNode(newNode);
}
  
```

Se l'albero è vuoto, non dovete fare altro che usare il nuovo nodo come radice. Diversamente, sapete che il nuovo nodo deve essere inserito da qualche parte tra i nodi e potete chiedere al nodo radice di effettuare l'inserimento. Quell'oggetto di tipo nodo invoca il metodo `addNode` della classe `Node`, che verifica se il nuovo oggetto è minore dell'oggetto immagazzinato nel nodo stesso. Se è così, l'elemento viene inserito nel sottoalbero sinistro del nodo, altrimenti viene inserito nel suo sottoalbero destro:

```

class Node
{
  
```

```

    .
    .
    public void addNode(Node newNode)
    {
        int comp = newNode.data.compareTo(data);
        if (comp < 0)
        {
            if (left == null) left = newNode;
            else left.addNode(newNode);
        }
        else if (comp > 0)
        {
            if (right == null) right = newNode;
            else right.addNode(newNode);
        }
    }
    .
}

```

Proviamo a seguire passo dopo passo le invocazioni del metodo `addNode` mentre si inserisce “Romeo” nell’albero della Figura 8. La prima invocazione di `addNode` è

```
root.addNode(newNode)
```

Siccome `root` contiene “Juliet”, si confronta “Juliet” con “Romeo” e si scopre di dover invocare

```
root.right.addNode(newNode)
```

Il nodo `root.right` è “Tom”. Si confrontano di nuovo i valori dei dati (“Tom” e “Romeo”) e ci si accorge che adesso si deve scendere verso sinistra. Dal momento che `root.right.left` è `null`, si rende `root.right.left` uguale a `newNode` e l’inserimento è concluso (osservate la Figura 9).

Diversamente da una lista concatenata o da un array, ma in analogia con le tabelle hash, un albero binario non è dotato di *posizioni di inserimento*: non è possibile scegliere la posizione in cui inserire un elemento all’interno di un albero di ricerca binario, in quanto la struttura si *auto-organizza*, cioè ciascun elemento viene inserito nella posizione che gli compete.

Analizzeremo ora l’algoritmo di rimozione di un dato dall’albero. Per prima cosa, ovviamente, dobbiamo *trovare* il nodo da eliminare, cosa piuttosto semplice, vista la proprietà caratteristica di un albero di ricerca binario: si confronta il valore del dato da rimuovere con il valore del dato memorizzato nel nodo radice, proseguendo la ricerca nel suo sottoalbero di sinistra o di destra in relazione al fatto che, rispettivamente, il primo valore sia minore o maggiore del secondo valore.

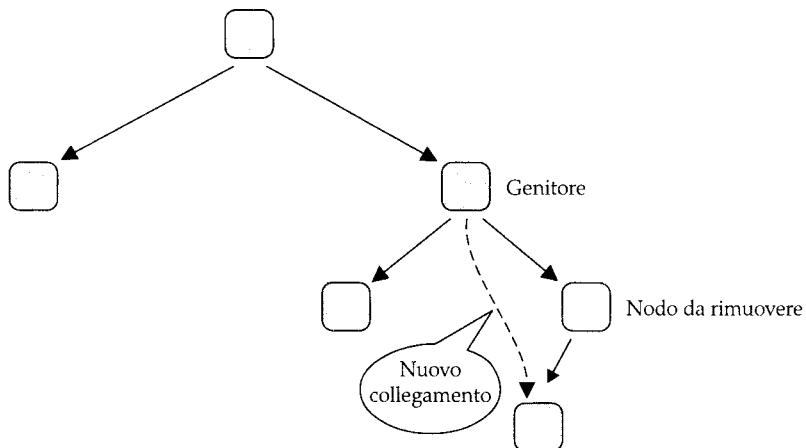
Immaginiamo di aver localizzato il nodo che deve essere eliminato dall’albero e consideriamo, per iniziare, un caso semplice: il nodo ha un unico figlio (osservate la Figura 10).

Per eliminare il nodo è sufficiente modificare, nel nodo genitore, il collegamento che punta al nodo stesso, in modo che vada invece a puntare al suo unico figlio.

Se il nodo da eliminare non ha figli, il corrispondente riferimento nel genitore viene semplicemente posto al valore `null`.

**Eliminando da un albero di ricerca binario un nodo avente un solo figlio, tale figlio prende il posto del nodo eliminato.**

**Figura 10**  
Eliminazione di un nodo  
avente un solo figlio



Il caso in cui il nodo da eliminare ha due figli è più complesso. Invece di eliminare il nodo, è più semplice sostituire il dato in esso memorizzato con il dato avente il valore successivo nella sequenza ordinata dei valori contenuti nell'albero, perché tale sostituzione preserva la proprietà fondamentale dell'albero di ricerca binario (in alternativa, come vedrete nell'Esercizio P15.21, si può usare l'elemento di valore maggiore presente nel sottoalbero sinistro, cioè il valore precedente nella sequenza).

Per identificare, nella sequenza ordinata dei valori contenuti nell'albero, il valore successivo a quello da eliminare, basta cercare il valore minimo presente nel sottoalbero destro del nodo che si dovrebbe eliminare, continuando, in tale sottoalbero destro, a seguire il collegamento che porta verso il figlio sinistro: una volta raggiunto un nodo privo di figlio sinistro, tale nodo è proprio quello che contiene il dato di valore minore all'interno dell'intero sottoalbero. Eliminate ora tale nodo, azione semplice perché può avere il solo figlio destro. Successivamente, memorizzate nel nodo originario (quello che doveva essere eliminato) il dato che era contenuto nel nodo appena eliminato. L'intera procedura è illustrata in dettaglio nella Figura 11 e, al termine di questo paragrafo, viene presentato il codice completo dell'algoritmo.

Al termine di questo paragrafo troverete il codice sorgente della classe `BinarySearchTree`, che contiene i metodi `add` e `remove` appena descritti, oltre al metodo `find`, che verifica se un determinato valore è presente nell'albero di ricerca binario, e al metodo `print`, di cui parleremo nel prossimo paragrafo.

Ora che avete visto la realizzazione di questa complessa struttura, potreste giustamente chiedervi se serve a qualcosa. Come in una lista, i nodi vengono creati uno alla volta. Nessun elemento esistente deve essere spostato quando si rimuove un elemento o se ne inserisce uno nuovo: questo è un vantaggio. La velocità dell'inserimento e della rimozione, però, dipende dalla forma dell'albero: sono operazioni veloci se l'albero è *bilanciato* (Figura 12).

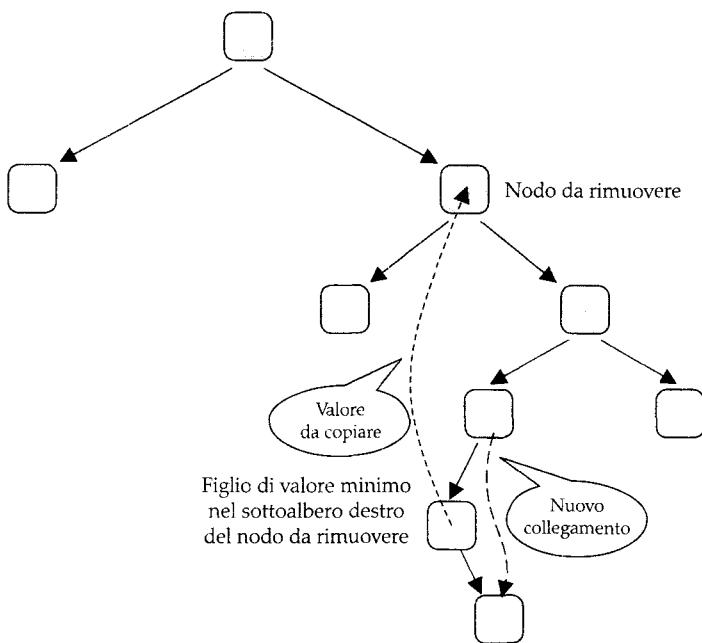
In un albero bilanciato, tutti i percorsi che vanno dalla radice a un nodo foglia (che è, ricordiamo, un nodo privo di figli) hanno approssimativamente la stessa lunghezza. Il numero di nodi che si trovano sul più lungo di tali percorsi è l'*altezza* dell'albero: gli alberi della Figura 12 hanno altezza 5.

Per eliminare da un albero di ricerca binario un nodo avente due figli, lo si sostituisce con il nodo di valore minimo presente nel suo sottoalbero destro.

In un albero bilanciato, tutti i percorsi che vanno dalla radice alle foglie hanno approssimativamente la stessa lunghezza.

In un albero, le operazioni di ricerca, inserimento e rimozione di un elemento richiedono un tempo proporzionale all'altezza dell'albero.

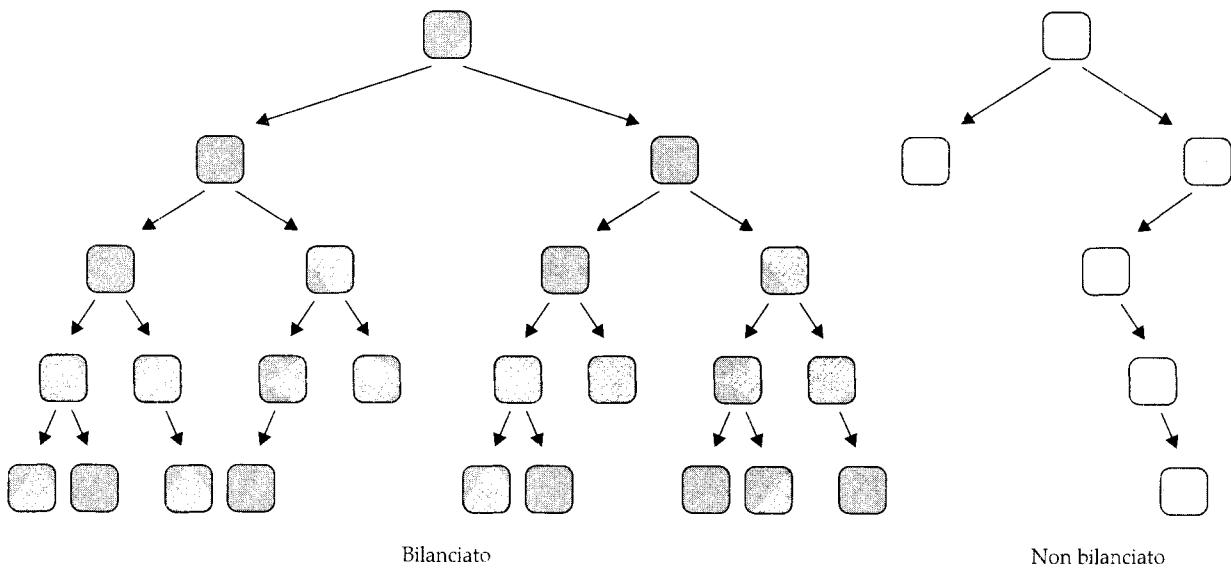
**Figura 11**  
Eliminazione di un nodo  
avente due figli



Dato che le operazioni di ricerca, inserimento e rimozione di un elemento elaborano i nodi che si trovano lungo un percorso che va dalla radice a una foglia, il loro tempo di esecuzione è proporzionale all'altezza dell'albero e non al numero totale di nodi presenti nell'albero stesso.

**Figura 12**  
Un albero bilanciato  
e uno non bilanciato

Un albero di altezza  $h$  può avere fino a  $n = 2^h - 1$  nodi. Ad esempio, un albero di altezza 4 completamente riempito ha  $1 + 2 + 4 + 8 = 15 = 2^4 - 1$  nodi. In altre parole, per un albero completamente riempito,  $h = \log_2(n + 1)$ . Nel caso di un albero bilanciato,



Se un albero di ricerca binario è bilanciato, le operazioni di ricerca, inserimento e rimozione di un elemento richiedono un tempo  $O(\log n)$ .

abbiamo ancora  $h \approx \log_2 n$ . Per esempio, l'altezza di un albero avente 1000 nodi è circa uguale a 10 (perché  $1024 = 2^{10}$ ), mentre un albero con un milione di nodi ha un'altezza approssimativamente pari a 20: in tale albero, si effettua la ricerca di un elemento in 20 passi, molto più velocemente della visita di un milione di nodi in una lista.

D'altro canto, se l'albero fosse *sbilanciato*, allora le operazioni potrebbero essere lente: nel caso peggiore, non più veloci di quelle di una lista concatenata.

Se i nuovi elementi sono abbastanza casuali, l'albero che ne risulta sarà probabilmente ben bilanciato. Se, però, gli elementi in arrivo sono già ordinati, allora l'albero che ne risulta è completamente sbilanciato: ciascun nuovo elemento viene inserito alla fine e ogni volta bisogna attraversare l'intero albero per trovare quell'estremità!

Gli alberi di ricerca binari vanno bene per dati casuali. Se avete il sospetto che i dati della vostra applicazione potrebbero essere ordinati o avere lunghe sequenze ordinate, non dovreste usare un albero di ricerca binario: esistono strutture ad albero più sofisticate, i cui metodi mantengono gli alberi costantemente bilanciati. Con queste strutture ad albero è possibile garantire che la ricerca, l'inserimento e l'eliminazione di elementi richieda un tempo  $O(\log(n))$ . Per realizzare insiemi e mappe, la libreria standard di Java usa *alberi rosso-nero* ("red-black trees"), un tipo speciale di alberi binari bilanciati.

### File ch15/tree/BinarySearchTree.java

```
/*
 * Questa classe implementa un albero di ricerca binario i cui
 * nodi contengono oggetti di tipo Comparable.
 */

public class BinarySearchTree
{
    private Node root;

    /**
     * Costruisce un albero vuoto.
     */
    public BinarySearchTree()
    {
        root = null;
    }

    /**
     * Inserisce un nuovo nodo nell'albero.
     * @param obj l'oggetto da inserire
     */
    public void add(Comparable obj)
    {
        Node newNode = new Node();
        newNode.data = obj;
        newNode.left = null;
        newNode.right = null;
        if (root == null) root = newNode;
        else root.addNode(newNode);
    }
}
```

```
/**  
 * Cerca un oggetto nell'albero.  
 * @param obj l'oggetto da cercare  
 * @return true se e solo se l'oggetto è presente nell'albero  
 */  
public boolean find(Comparable obj)  
{  
    Node current = root;  
    while (current != null)  
    {  
        int d = current.data.compareTo(obj);  
        if (d == 0) return true;  
        else if (d > 0) current = current.left;  
        else current = current.right;  
    }  
    return false;  
}  
  
/**  
 * Cerca di eliminare un oggetto dall'albero.  
 * Se l'oggetto non è presente non fa nulla.  
 * @param obj l'oggetto da eliminare  
 */  
public void remove(Comparable obj)  
{  
    // cerca il nodo da eliminare  
  
    Node toBeRemoved = root;  
    Node parent = null;  
    boolean found = false;  
    while (!found && toBeRemoved != null)  
    {  
        int d = toBeRemoved.data.compareTo(obj);  
        if (d == 0) found = true;  
        else  
        {  
            parent = toBeRemoved;  
            if (d > 0) toBeRemoved = toBeRemoved.left;  
            else toBeRemoved = toBeRemoved.right;  
        }  
    }  
  
    if (!found) return;  
  
    // toBeRemoved contiene obj  
  
    // se uno dei figli è vuoto, si usa l'altro  
  
    if (toBeRemoved.left == null || toBeRemoved.right == null)  
    {  
        Node newChild;  
        if (toBeRemoved.left == null)  
            newChild = toBeRemoved.right;  
        else  
            newChild = toBeRemoved.left;  
    }
```

```

        if (parent == null) // trovato in root
            root = newChild;
        else if (parent.left == toBeRemoved)
            parent.left = newChild;
        else
            parent.right = newChild;
        return;
    }

    // nessun sottoalbero è vuoto

    // cerca l'elemento minimo nel sottoalbero destro

    Node smallestParent = toBeRemoved;
    Node smallest = toBeRemoved.right;
    while (smallest.left != null)
    {
        smallestParent = smallest;
        smallest = smallest.left;
    }

    // smallest è il nodo che contiene l'elemento minimo
    // presente nel sottoalbero destro

    // copia il dato, scollega il figlio

    toBeRemoved.data = smallest.data;
    if (smallestParent == toBeRemoved)
        smallestParent.right = smallest.right;
    else
        smallestParent.left = smallest.right;
}

/**
 * Visualizza il contenuto dell'albero in successione ordinata.
 */
public void print()
{
    if (root != null)
        root.printNodes();
    System.out.println();
}

/**
 * Un nodo di un albero memorizza un dato e i
 * riferimenti a due nodi, figlio sinistro e figlio destro.
 */
class Node
{
    public Comparable data;
    public Node left;
    public Node right;

    /**
     * Inserisce un nuovo nodo come discendente di questo nodo.
     * @param newNode il nodo da inserire

```

```

    */
public void addNode(Node newNode)
{
    int comp = newNode.data.compareTo(data);
    if (comp < 0)
    {
        if (left == null) left = newNode;
        else left.addNode(newNode);
    }
    else if (comp > 0)
    {
        if (right == null) right = newNode;
        else right.addNode(newNode);
    }
}

/**
 * Visualizza questo nodo e tutti i suoi discendenti
 * in successione ordinata.
 */
public void printNodes()
{
    if (left != null)
        left.printNodes();
    System.out.print(data + " ");
    if (right != null)
        right.printNodes();
}
}
}

```



### Auto-valutazione

11. Quale differenza esiste tra un albero, un albero binario e un albero binario bilanciato?
12. Fornite un esempio di stringa che, inserita nell'albero rappresentato nella Figura 9, si venga a trovare nel figlio destro del nodo che contiene la stringa Romeo.

## 15.6 Visita di un albero

Adesso che i dati sono stati inseriti nell'albero, come lo potete usare? Risulta sorprendentemente semplice, ad esempio, stampare tutti gli elementi in successione ordinata. Voi sapete che tutti i dati nel sottoalbero sinistro di qualsiasi nodo devono venire prima del dato contenuto nel nodo e prima di tutti i dati contenuti nel sottoalbero destro, per cui il seguente algoritmo stamperà gli elementi in successione ordinata:

1. Stampa il sottoalbero sinistro.
2. Stampa il dato.
3. Stampa il sottoalbero destro.

Facciamo una prova con l'albero della Figura 9. L'algoritmo ci dice di

1. Stampare il sottoalbero sinistro di “Juliet”, cioè “Diana” e i suoi discendenti.
2. Stampare “Juliet”.
3. Stampare il sottoalbero destro di “Juliet”, cioè “Tom” e i suoi discendenti.

Come fate a stampare il sottoalbero che ha come radice “Diana”?

1. Stampate il sottoalbero sinistro di “Diana”: non c’è niente da stampare.
2. Stampate “Diana”.
3. Stampate il sottoalbero destro di “Diana”, cioè “Harry”.

Quindi, il sottoalbero sinistro di “Juliet” viene stampato come

Diana Harry

Il sottoalbero destro di “Juliet” è il sottoalbero che ha come radice “Tom”. Come viene stampato? Ancora una volta, utilizzando lo stesso algoritmo:

1. Stampate il sottoalbero sinistro di “Tom”, cioè “Romeo”.
2. Stampate “Tom”.
3. Stampate il sottoalbero destro di “Tom”: non c’è niente da stampare.

Quindi, il sottoalbero destro di “Juliet” viene stampato come

Romeo Tom

Ora mettete tutto assieme: il sottoalbero sinistro, “Juliet” e il sottoalbero destro:

Diana Harry Juliet Romeo Tom

L’albero viene stampato in successione ordinata.

Realizziamo ora il metodo `print`, per il quale serve un metodo ausiliario, `printNodes`, nella classe `Node`:

```
class Node
{
    ...
    public void printNodes()
    {
        if (left != null)
            left.printNodes();
        System.out.print(data + " ");

        if (right != null)
            right.printNodes();
    }
    ...
}
```

Per stampare l’intero albero, iniziate questo processo ricorsivo dalla radice, invocando questo metodo della classe `BinarySearchTree`:

```

public class BinarySearchTree
{
    . .
    public void print()
    {
        if (root != null)
            root.printNodes();
        System.out.println();
    }
    . .
}

```

Per visitare tutti gli elementi di un albero, si visita la radice e, ricorsivamente, si visitano i sottoalberi. Distinguiamo tre tipi di visite: **in ordine simmetrico**, **in ordine anticipato** e **in ordine posticipato**.

Questa strategia viene chiamata *visita in ordine simmetrico* (“*inorder traversal*”, che prevede la visita del sottoalbero sinistro, della radice e del sottoalbero destro), ma esistono altre due importanti strategie di attraversamento di alberi, denominate *visita in ordine anticipato* (“*preorder traversal*”) e *visita in ordine posticipato* (“*postorder traversal*”).

Nell’attraversamento in ordine anticipato:

- Si visita la radice
- Si visita il suo sottoalbero sinistro
- Si visita il suo sottoalbero destro

Nell’attraversamento in ordine posticipato:

- Si visita il sottoalbero sinistro della radice
- Si visita il sottoalbero destro della radice
- Si visita la radice

Queste due strategie di visita non stampano in ordine i valori contenuti nell’albero, ma sono importanti in altre applicazioni relative agli alberi binari, di cui forniamo ora un esempio.

Nel Capitolo 12 abbiamo presentato un algoritmo per l’analisi sintattica di espressioni aritmetiche di questo tipo:

$$(3 + 4) * 5$$

$$3 + 4 * 5$$

Spesso queste espressioni vengono rappresentate mediante alberi, come si può vedere nella Figura 13. Se tutti gli operatori avessero due operandi, l’albero risultante sarebbe binario, con le foglie che memorizzano numeri e i nodi interni che memorizzano operatori.

Note che tali alberi che rappresentano espressioni descrivono anche l’ordine in cui vengono applicati gli operatori, ordine che diviene evidente quando si effettua l’attraversamento in ordine posticipato dell’albero di espressione.

Per il primo albero si ha:

$$3 \ 4 \ + \ 5 \ *$$

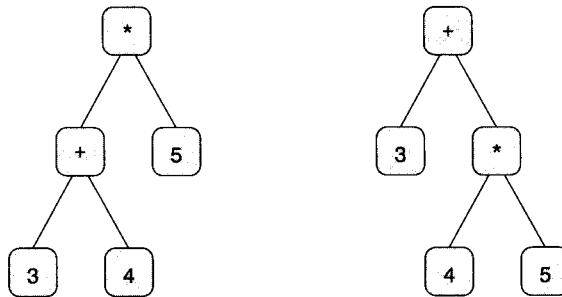
mentre per il secondo si ottiene:

$$3 \ 4 \ 5 \ * \ +$$

L’attraversamento in ordine posticipato di un albero di espressione fornisce le istruzioni necessarie per valutare l’espressione con una calcolatrice con funzionamento *a stack*.

**Figura 13**

Alberi che rappresentano un'espressione



Queste sequenze possono essere interpretate come espressioni in “notazione polacca inversa” (RPN, si vedano gli Argomenti avanzati 14.1), oppure, in modo equivalente, come istruzioni per una calcolatrice con funzionamento *a stack* (si vedano gli Esempi completi 14.1).



### Auto-valutazione

13. Quali sono gli attraversamenti in ordine simmetrico dei due alberi della Figura 13?
14. Gli alberi della Figura 13 sono alberi di ricerca binari?

## 15.7 Code prioritarie

Nel Paragrafo 14.4 avete visto due tipi di dati astratti molto comuni: pile e code. Un altro importante tipo di dato astratto, la *coda prioritaria*, funge da contenitore di elementi, a ciascuno dei quali viene assegnata una *priorità*. Un tipico esempio di coda prioritaria è un insieme di richieste di lavori da svolgere, alcuni dei quali possono essere più urgenti di altri. Diversamente da una coda normale, la coda prioritaria non realizza una strategia di tipo “il primo entrato è il primo a uscire” (FIFO), ma gli elementi vengono estratti in base alla loro priorità: in altre parole, si possono inserire nuovi elementi nella coda in qualsiasi ordine, ma, quando viene rimosso un elemento dalla coda, tale elemento sarà quello di priorità massima.

Per convenzione, solitamente si assegnano valori più bassi alle priorità più alte, con la priorità di valore 1 che indica la priorità massima: la coda prioritaria, quindi, elimina sempre dalla coda l’elemento *minimo*.

Considerate, ad esempio, il codice seguente:

```

PriorityQueue<WorkOrder> q = new PriorityQueue<WorkOrder>();
q.add(new WorkOrder(3, "Shampoo carpets"));
q.add(new WorkOrder(1, "Fix overflowing sink"));
q.add(new WorkOrder(2, "Order cleaning supplies"));
  
```

Quando si invoca per la prima volta `q.remove()`, viene estratto dalla coda l’ordine di lavoro con priorità 1. La successiva invocazione di `q.remove()` estrae l’ordine di lavoro di priorità più elevata tra quelli rimasti nella coda, che nel nostro esempio è l’ordine di lavoro con priorità 2.

La libreria standard di Java mette a disposizione una classe `PriorityQueue` che è

**Quando si elimina un elemento da una coda prioritaria viene rimosso l’elemento con la priorità più elevata.**

pronta all'uso, mentre nel seguito di questo capitolo imparerete a scriverne una vostra realizzazione.

Ricordatevi che la coda prioritaria è un tipo di dati *astratto*: non sapete come vengono organizzati gli elementi all'interno di una coda prioritaria ed esistono diverse strutture dati concrete che possono essere utilizzate per realizzare code prioritarie.

Una particolare realizzazione, però, viene immediatamente alla mente: memorizzare semplicemente gli elementi in una lista concatenata, aggiungendo nuovi elementi all'inizio della lista; il metodo `remove`, poi, scandisce la lista concatenata ed elimina l'elemento con la priorità più elevata. Con tale realizzazione, l'aggiunta di un nuovo elemento è un'operazione veloce, mentre la rimozione è lenta.

Un'altra strategia di realizzazione consiste nel mantenere ordinati gli elementi, ad esempio in un albero di ricerca binario. In tal caso è poi agevole localizzare ed eliminare l'elemento a priorità massima, anche se un'altra struttura dati, chiamata “heap”, è ancora più adatta per realizzare code prioritarie.

## 15.8 Heap

**Uno heap è un albero quasi completo  
in cui ogni nodo ha un valore  
non superiore a quelli dei propri  
discendenti.**

Uno *heap* (“mucchio”, chiamato anche, per maggior chiarezza, *min-heap*) è un albero binario dotato di due speciali proprietà.

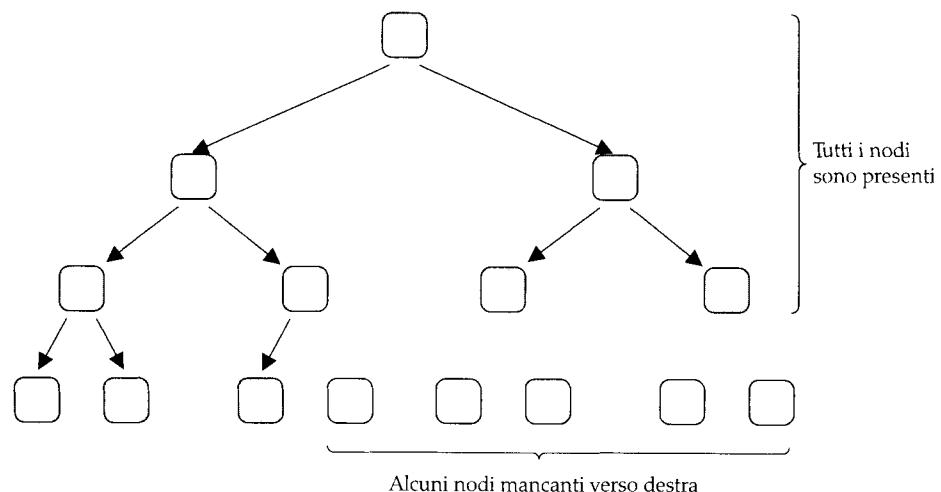
1. Uno heap è *quasi completo*: a ogni livello sono presenti tutti i nodi possibili, tranne al livello più basso dell'albero, dove può mancare qualche nodo nella parte destra (si osservi la Figura 14).
2. L'albero soddisfa la “proprietà di heap”: ogni nodo memorizza un valore che non è superiore ai valori memorizzati nei suoi discendenti (si osservi la Figura 15).

È facile verificare che la proprietà di heap garantisce che nella radice sia memorizzato il valore minimo.

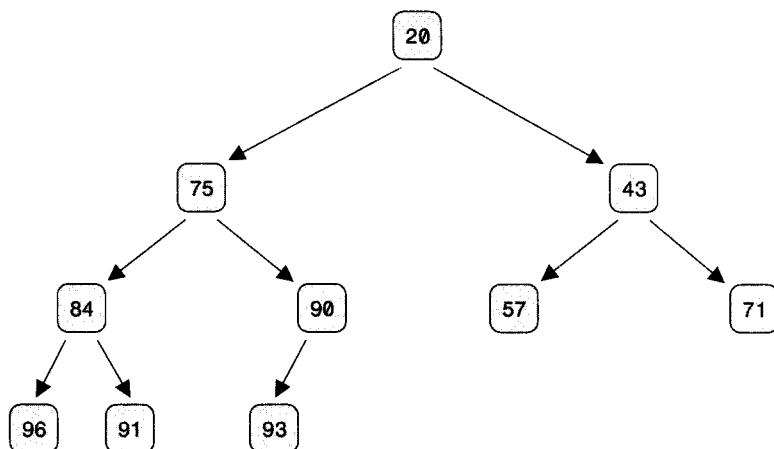
A prima vista, uno heap sembra simile a un albero di ricerca binario, ma ci sono due differenze importanti.

**Figura 14**

Un albero quasi completo



**Figura 15**  
Uno heap

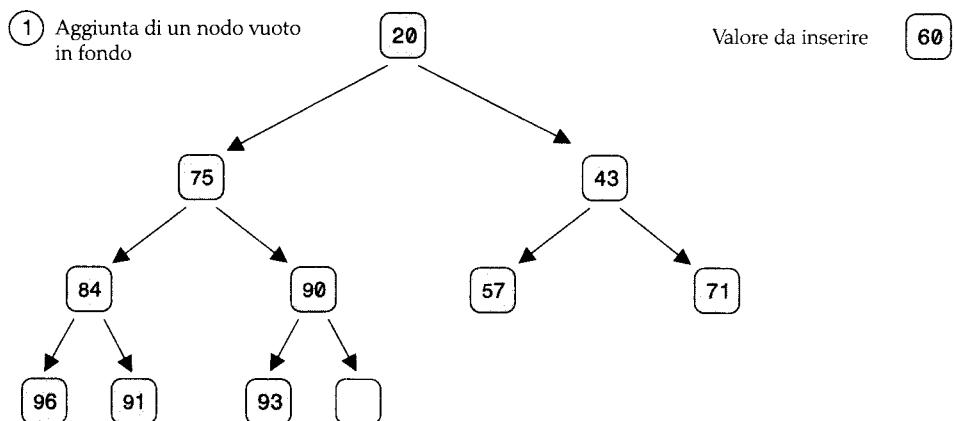


1. La forma di uno heap è molto regolare, mentre gli alberi di ricerca binari possono avere forme arbitrarie.
2. In uno heap, sia il sottoalbero di sinistra che il sottoalbero di destra contengono elementi di valore maggiore o uguale all'elemento radice, mentre, al contrario, in un albero di ricerca binario gli elementi minori sono memorizzati nel sottoalbero sinistro e gli elementi maggiori sono memorizzati nel sottoalbero destro.

Supponete di avere uno heap e di volervi inserire un nuovo elemento: ovviamente, dopo l'inserimento la proprietà di heap deve essere ancora vera. Il seguente algoritmo effettua l'inserimento, come si può vedere nella Figura 16.

1. Per prima cosa, aggiungete alla fine dello heap un nodo vuoto.
2. Successivamente, se il genitore del nodo vuoto ha un valore maggiore dell'elemento da inserire, "declassatelo", cioè spostate nel nodo vuoto il valore contenuto nel genitore, rendendo così vuoto il nodo genitore (apparentemente la "lacuna" si sposta verso l'alto). Ripetete il declassamento finché il genitore del nodo vuoto ha un valore maggiore dell'elemento da inserire, come si può vedere nella terza fase della Figura 16.

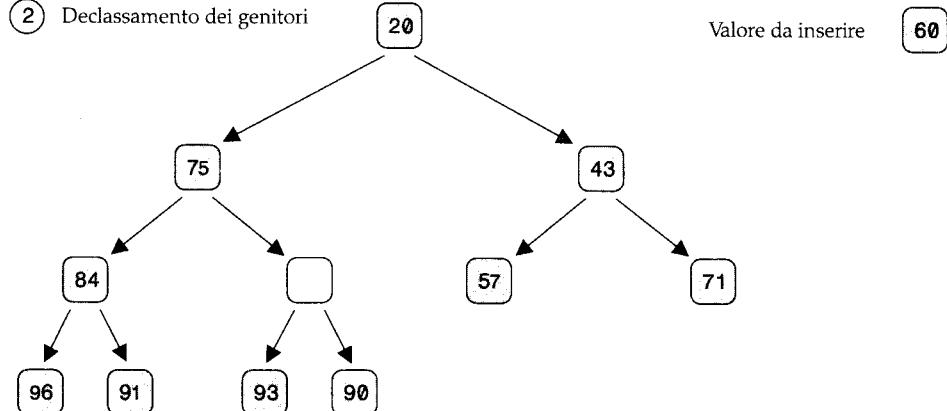
**Figura 16**  
Inserimento  
di un elemento  
in uno heap



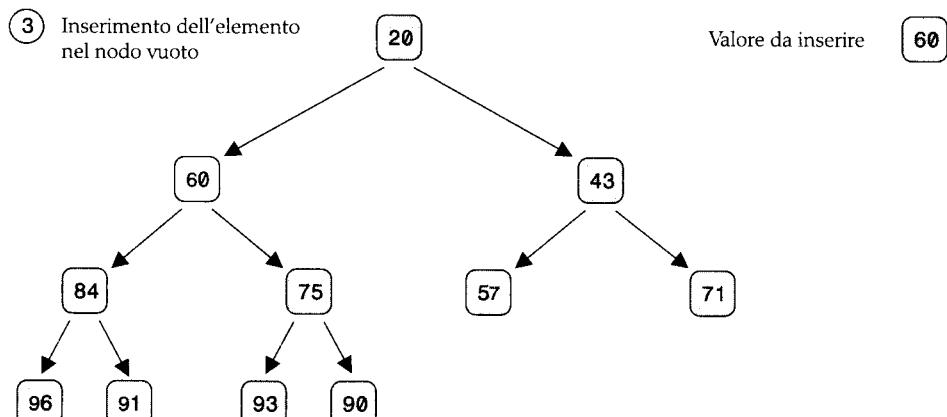
**Figura 16**

(continua)

② Declassamento dei genitori



③ Inserimento dell'elemento nel nodo vuoto

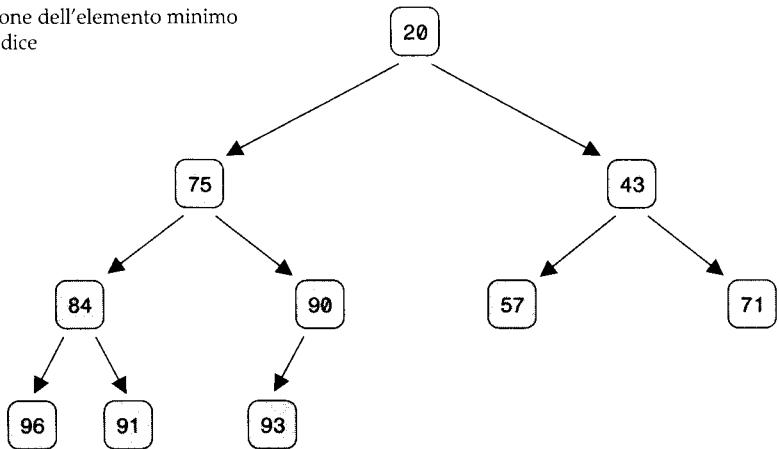


3. A questo punto il nodo vuoto si trova nella radice, oppure il genitore del nodo vuoto ha un valore inferiore all'elemento da inserire. Inserite l'elemento nel nodo vuoto.

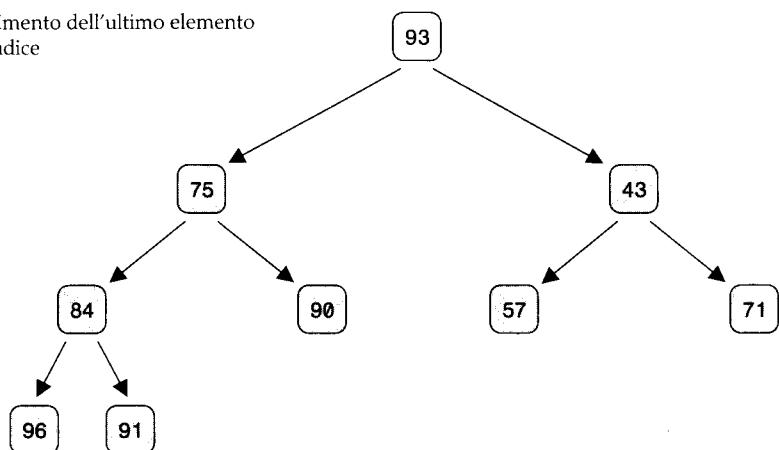
**Figura 17**

Rimozione dell'elemento minimo  
dalla radice

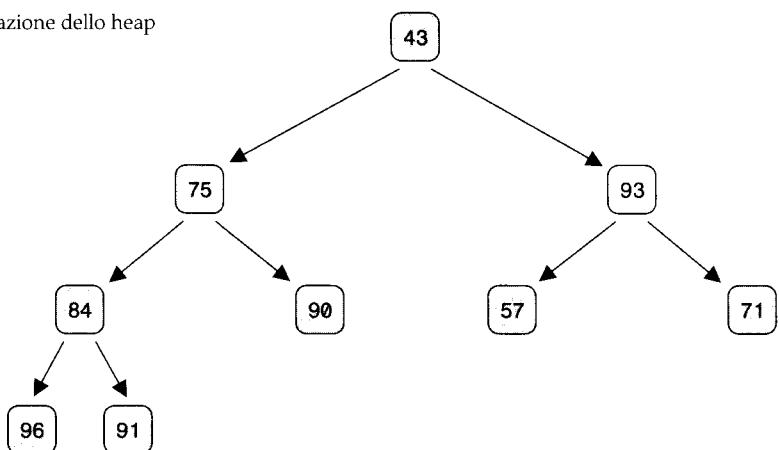
- ① Rimozione dell'elemento minimo  
dalla radice



- ② Trasferimento dell'ultimo elemento  
nella radice

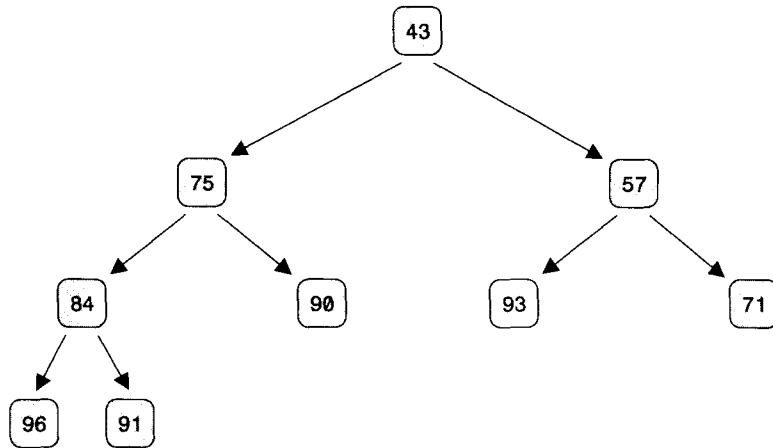


- ③ Sistemazione dello heap



**Figura 17**

(continua)



Non esamineremo l'algoritmo che elimina dallo heap un nodo qualsiasi. L'unico nodo che elimineremo è il nodo radice, che contiene il valore minimo tra tutti i valori presenti nello heap. La Figura 17 mostra tale algoritmo in azione.

1. Eliminate il valore presente nel nodo radice.
2. Trasferite nella radice il valore presente nell'ultimo nodo dello heap ed eliminate tale nodo. A questo punto può darsi che il nodo radice non rispetti la proprietà di heap, perché uno dei suoi figli (o entrambi) può avere un valore inferiore.
3. "Promuovete" il figlio della radice che ha il valore minore, tra i due, come si può vedere nella Figura 17: ora il nodo radice rispetta nuovamente la proprietà di heap. Ripetete questo procedimento con il figlio che è stato appena declassato, cioè promuovete il suo figlio di valore minore; continuate fino a quando il figlio declassato si trova ad avere figli di valore non inferiore a sé. A questo punto la proprietà di heap è nuovamente soddisfatta. Questo procedimento viene chiamato "sistematizzazione dello heap" (*fixing the heap*).

L'inserimento e la rimozione di elementi in uno heap sono operazioni molto efficienti, per via della forma bilanciata dello heap. Le operazioni di inserimento e rimozione visitano al massimo  $h$  nodi, dove  $h$  è l'altezza dell'albero. Uno heap di altezza  $h$  contiene almeno  $2^{h-1}$  elementi, ma meno di  $2^h$  elementi. In altre parole, se  $n$  è il numero di elementi, allora:

$$2^{h-1} \leq n < 2^h$$

oppure, equivalentemente:

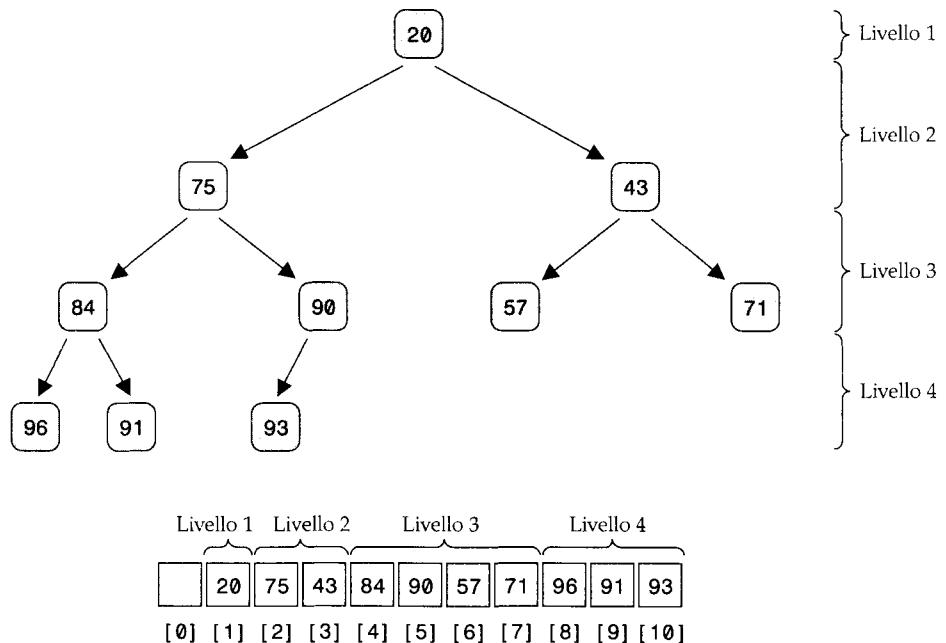
$$h - 1 \leq \log_2(n) < h$$

In uno heap, l'inserimento e la rimozione di un elemento sono operazioni  $O(\log n)$ .

Ciò dimostra che le operazioni di inserimento e rimozione in uno heap avente  $n$  elementi richiedono  $O(\log n)$  passi.

Confrontate questo risultato con la situazione che abbiamo visto in relazione agli alberi di ricerca binari. Quando un albero di ricerca binario è sbilanciato, degenera in una lista concatenata, per cui le operazioni di inserimento e rimozione, nel caso peggiore, sono  $O(n)$ .

**Figura 18**  
Memorizzazione  
di uno heap all'interno  
di un array



La disposizione regolare dei nodi  
di uno heap rende possibile  
una loro efficiente memorizzazione  
in un array.

Gli heap hanno anche un altro grande vantaggio: per effetto della disposizione regolare dei nodi, è facile memorizzarne i valori in un array o in un vettore. Si memorizza per prima cosa il primo livello dell'albero, poi il secondo e così via, come si può vedere nella Figura 18, lasciando, per comodità, vuota la cella dell'array di indice 0. Di conseguenza, i nodi figli del nodo di indice  $i$  hanno indici  $2 \times i$  e  $2 \times i + 1$ , mentre il nodo genitore del nodo di indice  $i$  ha indice  $i / 2$ . Ad esempio, come potete vedere dalla Figura 18, i figli del nodo 4 sono i nodi 8 e 9, mentre il suo genitore è il nodo 2.

La memorizzazione dei valori di uno heap in un array può non essere intuitiva, ma è molto efficiente: non c'è più bisogno di creare i singoli nodi o di memorizzare i collegamenti ai nodi figli, dal momento che le posizioni dei figli e dei genitori possono essere determinate con calcoli molto semplici.

Il programma presentato al termine di questo paragrafo contiene la realizzazione di uno heap. Per maggior chiarezza, i calcoli degli indici che rappresentano le posizioni di genitori e figli di un nodo vengono eseguiti in metodi ausiliari, `getParentIndex`, `getLeftChildIndex` e `getRightChildIndex`. Per ottenere una maggior efficienza, si potrebbero evitare le ripetute invocazioni di tali metodi usando direttamente le espressioni `index / 2`, `2 * index` e `2 * index + 1`.

In questo paragrafo abbiamo organizzato lo heap in modo che l'elemento di valore minimo sia memorizzato nella radice, ma è anche possibile memorizzarvi l'elemento massimo, semplicemente invertendo tutti i confronti eseguiti dall'algoritmo che costruisce lo heap. In caso vi sia possibilità di fraintendimento, è meglio riferirsi alle due diverse organizzazioni dello heap come *min-heap* e *max-heap*.

Il programma di prova mostra come si possa usare un min-heap per realizzare una coda prioritaria.

**File ch15/pqueue/MinHeap.java**

```
import java.util.*;  
  
/**  
 * Classe che realizza uno heap.  
 */  
public class MinHeap  
{  
    private ArrayList<Comparable> elements;  
  
    /**  
     * Costruisce uno heap vuoto.  
     */  
    public MinHeap()  
    {  
        elements = new ArrayList<Comparable>();  
        elements.add(null);  
    }  
  
    /**  
     * Aggiunge un nuovo elemento allo heap.  
     * @param newElement l'elemento da aggiungere  
     */  
    public void add(Comparable newElement)  
    {  
        // aggiunge una nuova foglia  
        elements.add(null);  
        int index = elements.size() - 1;  
  
        // declassa i genitori che hanno valore maggiore  
        // del nuovo elemento  
        while (index > 1  
              && getParent(index).compareTo(newElement) > 0)  
        {  
            elements.set(index, getParent(index));  
            index = getParentIndex(index);  
        }  
  
        // memorizza il nuovo elemento nel nodo vuoto  
        elements.set(index, newElement);  
    }  
  
    /**  
     * Restituisce l'elemento minimo presente nello heap.  
     * @return l'elemento minimo  
     */  
    public Comparable peek()  
    {  
        return elements.get(1);  
    }  
  
    /**  
     * Elimina dallo heap l'elemento minimo.  
     * @return l'elemento minimo  
     */
```

```

        */
    public Comparable remove()
    {
        Comparable minimum = elements.get(1);

        // elimina l'ultimo elemento
        int lastIndex = elements.size() - 1;
        Comparable last = elements.remove(lastIndex);

        if (lastIndex > 1)
        {
            elements.set(1, last);
            fixHeap();
        }

        return minimum;
    }

    /**
     * Ripristina la proprietà di heap, a condizione che
     * soltanto il nodo radice la violi.
     */
    private void fixHeap()
    {
        Comparable root = elements.get(1);

        int lastIndex = elements.size() - 1;

        // promuove i figli della radice rimossa finché
        // sono minori dell'ultimo elemento
        int index = 1;
        boolean more = true;
        while (more)
        {
            int childIndex = getLeftChildIndex(index);
            if (childIndex <= lastIndex)
            {
                // identifica il figlio minore

                // considera prima il figlio sinistro
                Comparable child = getLeftChild(index);

                // usa invece il figlio destro se è minore
                if (getRightChildIndex(index) <= lastIndex
                    && getRightChild(index).compareTo(child) < 0)
                {
                    childIndex = getRightChildIndex(index);
                    child = getRightChild(index);
                }

                // verifica se il figlio maggiore è minore della radice
                if (child.compareTo(root) < 0)
                {
                    // promuove il figlio
                    elements.set(index, child);
                    index = childIndex;
                }
            }
        }
    }
}

```

```
        }
    else
    {
        // la radice è minore di entrambi i figli
        more = false;
    }
}
else
{
    // non ci sono figli
    more = false;
}
}

// memorizza nel nodo vuoto l'elemento presente
// nella radice
elements.set(index, root);
}

/**
 * Restituisce il numero di elementi presenti nello heap.
 */
public int size()
{
    return elements.size() - 1;
}

/**
 * Restituisce l'indice del figlio sinistro.
 * @param index l'indice di un nodo dello heap
 * @return l'indice del figlio sinistro del nodo dato
 */
private static int getLeftChildIndex(int index)
{
    return 2 * index;
}

/**
 * Restituisce l'indice del figlio destro.
 * @param index l'indice di un nodo dello heap
 * @return l'indice del figlio destro del nodo dato
 */
private static int getRightChildIndex(int index)
{
    return 2 * index + 1;
}

/**
 * Restituisce l'indice del genitore.
 * @param index l'indice di un nodo dello heap
 * @return l'indice del genitore del nodo dato
 */
private static int getParentIndex(int index)
{
    return index / 2;
}
```

```

    /**
     * Restituisce il valore del figlio sinistro.
     * @param index l'indice di un nodo dello heap
     * @return il valore del figlio sinistro del nodo dato
    */
    private Comparable getLeftChild(int index)
    {
        return elements.get(2 * index);
    }

    /**
     * Restituisce il valore del figlio destro.
     * @param index l'indice di un nodo dello heap
     * @return il valore del figlio destro del nodo dato
    */
    private Comparable getRightChild(int index)
    {
        return elements.get(2 * index + 1);
    }

    /**
     * Restituisce il valore del genitore.
     * @param index l'indice di un nodo dello heap
     * @return il valore del genitore del nodo dato
    */
    private Comparable getParent(int index)
    {
        return elements.get(index / 2);
    }
}

```

### File ch15/pqueue/WorkOrder.java

```

    /**
     * Questa classe incapsula un ordine di lavoro
     * avente una priorità.
    */
    public class WorkOrder implements Comparable
    {
        private int priority;
        private String description;

        /**
         * Costruisce un ordine di lavoro con una data priorità
         * e descrizione.
         * @param aPriority la priorità dell'ordine di lavoro
         * @param aDescription la descrizione dell'ordine di lavoro
        */
        public WorkOrder(int aPriority, String aDescription)
        {
            priority = aPriority;
            description = aDescription;
        }

        public String toString()
        {

```

```

        return "priority=" + priority + ", description=" + description;
    }

    public int compareTo(Object otherObject)
    {
        WorkOrder other = (WorkOrder) otherObject;
        if (priority < other.priority) return -1;
        if (priority > other.priority) return 1;
        return 0;
    }
}

```

### File ch15/pqueue/HeapDemo.java

```

/*
 * Questo programma illustra l'uso di uno heap
 * come coda prioritaria.
 */
public class HeapDemo
{
    public static void main(String[] args)
    {
        MinHeap q = new MinHeap();
        q.add(new WorkOrder(3, "Shampoo carpets"));
        q.add(new WorkOrder(7, "Empty trash"));
        q.add(new WorkOrder(8, "Water plants"));
        q.add(new WorkOrder(10, "Remove pencil sharpener shavings"));
        q.add(new WorkOrder(6, "Replace light bulb"));
        q.add(new WorkOrder(1, "Fix broken sink"));
        q.add(new WorkOrder(9, "Clean coffee maker"));
        q.add(new WorkOrder(2, "Order cleaning supplies"));

        while (q.size() > 0)
            System.out.println(q.remove());
    }
}

```

### Esecuzione del programma

```

priority=1, description=Fix broken sink
priority=2, description=Order cleaning supplies
priority=3, description=Shampoo carpets
priority=6, description=Replace light bulb
priority=7, description=Empty trash
priority=8, description=Water plants
priority=9, description=Clean coffee maker
priority=10, description=Remove pencil sharpener shavings

```



### Auto-valutazione

15. Il software che controlla gli eventi di un'interfaccia utente conserva gli eventi in una struttura dati: ogni volta che avviene un evento, come il movimento del mouse o una richiesta di ridisegno, l'evento viene aggiunto alla struttura dati; gli eventi vengono poi da essa estratti secondo la propria importanza. Quale tipo di dati astratto è appropriato per tale applicazione?

16. Potremmo memorizzare un albero di ricerca binario in un array, in modo da localizzare rapidamente i figli del nodo di indice `index` esaminando le celle dell'array identificate dalle espressioni  $2 * \text{index}$  e  $2 * \text{index} + 1$ ?

## 15.9 L'algoritmo Heapsort

L'algoritmo *heapsort* è basato sull'inserimento di elementi in uno heap e sulla loro successiva rimozione ordinata.

L'algoritmo *heapsort* ha prestazioni  $O(n \log n)$ .

Gli heap non sono utili solamente per realizzare code prioritarie, ma consentono anche la realizzazione di un algoritmo di ordinamento molto efficiente, denominato *heapsort*, che nella sua forma più semplice funziona in questo modo: dapprima tutti gli elementi da ordinare vengono inseriti in uno heap, poi se ne estrae ripetutamente l'elemento minimo fino a vuotarlo.

Questo algoritmo ha prestazioni  $O(n \log n)$ : ciascun inserimento e rimozione è un'operazione  $O(\log n)$  e tali due operazioni vengono ripetute  $n$  volte ciascuna, una volta per ogni elemento della sequenza che deve essere ordinata.

L'algoritmo può essere reso un po' più efficiente. Invece di inserire un elemento per volta, iniziamo con una sequenza di valori inseriti in un array, che, ovviamente, non rappresenta uno heap. Useremo, come parte di questo algoritmo, la procedura di "sistematizzazione dello heap" vista nel paragrafo precedente come parte dell'algoritmo di rimozione dell'elemento minimo. La "sistematizzazione dello heap" opera su un albero binario i cui alberi figli sono heap ma la cui radice contiene un valore che può essere maggiore di qualche suo discendente. La procedura trasforma l'albero in uno heap "promuovendo" ripetutamente il figlio di valore minimo, spostando così nella sua posizione corretta il valore presente originariamente nella radice.

Ovviamente, non è possibile applicare semplicemente tale procedura alla sequenza iniziale di valori non ordinati, perché è improbabile che gli alberi figli della radice siano degli heap: possiamo, però, trasformare in heap dapprima dei piccoli sottoalberi, per poi passare ad alberi più grandi. Dato che gli alberi di dimensione 1 sono automaticamente degli heap, possiamo iniziare la procedura con i sottoalberi le cui radici si trovano nel penultimo livello dell'albero.

L'algoritmo di ordinamento usa un metodo `fixHeap` generalizzato che trasforma in heap un sottoalbero avente radice in un elemento di indice dato, `rootIndex`:

```
void fixHeap(int rootIndex, int lastIndex)
```

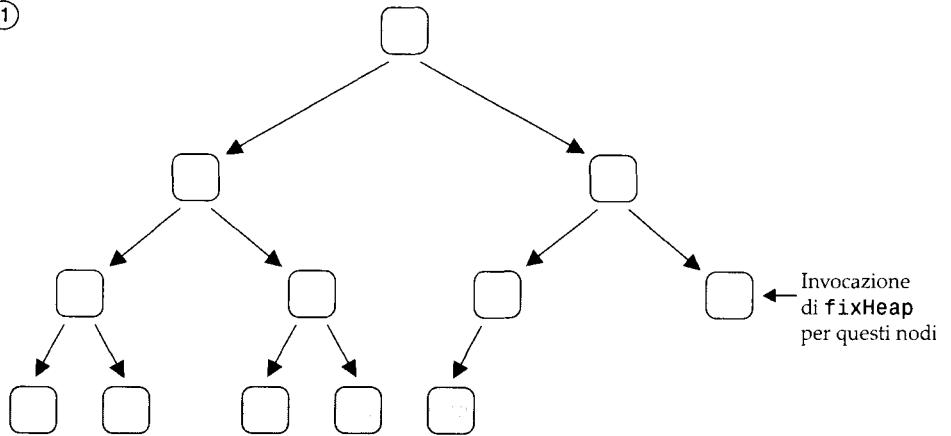
Tra i parametri del metodo, `lastIndex` è l'indice dell'ultimo nodo presente nell'intero albero.

Il metodo `fixHeap` deve essere invocato su tutti i sottoalberi le cui radici si trovano nel penultimo livello; successivamente, vengono sistematati i sottoalberi le cui radici si trovano nel livello immediatamente superiore a quello già sistemato, e così via. Infine, la procedura di sistematizzazione viene applicata al nodo radice e l'intero albero viene così trasformato in heap (Figura 19).

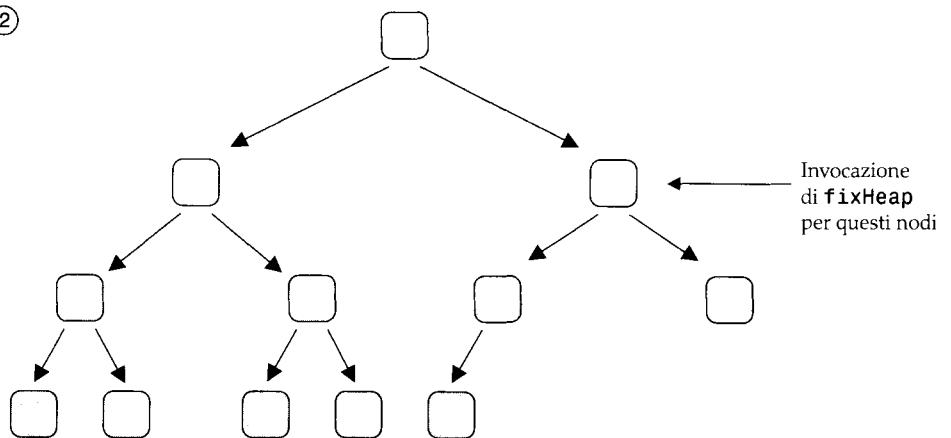
Tale procedura ripetitiva è semplice da realizzare con un programma. Si inizia con l'ultimo nodo del penultimo livello e si procede verso sinistra, poi si passa al livello superiore. In questo modo, i valori degli indici dei nodi vanno semplicemente all'indietro, partendo dall'indice dell'ultimo nodo per finire con l'indice della radice.

**Figura 19**  
Trasformazione  
di un albero in uno heap

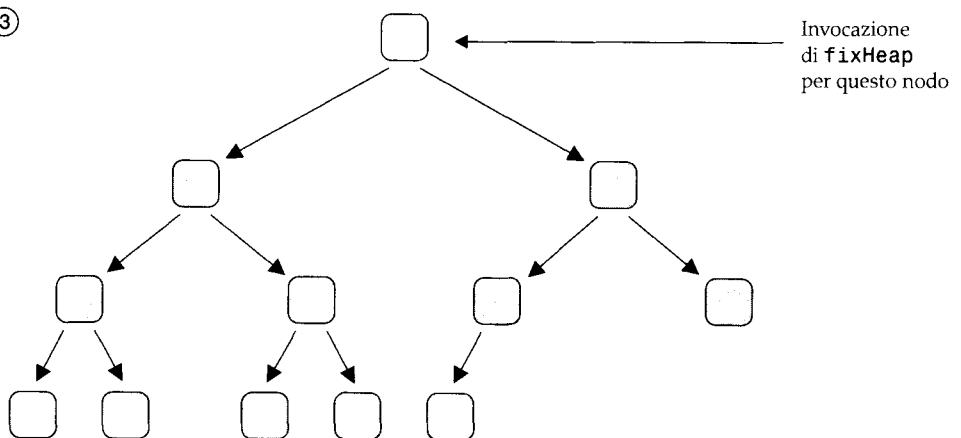
(1)



(2)



(3)



```

int n = a.length - 1;
for (int i = (n - 1) / 2; i >= 0; i--)
    fixHeap(i, n);

```

Si può dimostrare che questa procedura trasforma un array di contenuto arbitrario in uno heap in  $O(n)$  passi.

Noteate che il ciclo termina con l'indice 0. Quando operiamo su un array, non possiamo concederci il lusso di ignorare il valore di indice 0, per cui dobbiamo considerare tale valore come radice e modificare le formule che calcolano i valori degli indici dei figli e del genitore di un nodo.

Dopo aver trasformato l'array in heap, ne eliminiamo ripetutamente l'elemento radice. Dal paragrafo precedente, ricordate che l'eliminazione della radice si ottiene sostituendola con l'ultimo elemento presente nell'albero, invocando poi il metodo `fixHeap`. Dato che invochiamo  $n$  volte il metodo `fixHeap`, che ha prestazioni  $O(\log n)$ , l'intera procedura richiede  $O(n \log n)$  passi.

Invece di spostare l'elemento radice in un array distinto, *scambieremo* l'elemento radice con l'ultimo elemento dell'albero, riducendo conseguentemente la lunghezza "logica" dell'albero. La radice eliminata viene a trovarsi nell'ultima posizione dell'array, che non è più utilizzata dallo heap: così facendo, siamo in grado di usare il medesimo array sia per contenere lo heap (che, a ogni passo, diventa più piccolo) sia la sequenza ordinata (che, al contrario, diventa sempre più lunga).

```

while (n > 0)
{
    swap(0, n);
    n--;
    fixHeap(0, n);
}

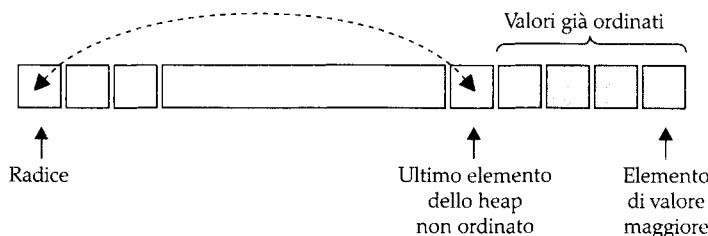
```

C'è soltanto un piccolo inconveniente: usando un min-heap, la sequenza ordinata viene memorizzata al contrario, con l'elemento più piccolo che viene a trovarsi alla fine dell'array. Potremmo invertire la sequenza al termine dell'ordinamento, ma è più semplice usare un max-heap nell'algoritmo heapsort. Con tale modifica, durante il primo passo dell'algoritmo il valore maggiore viene inserito nell'ultima posizione dell'array; al passo successivo, il secondo valore più elevato viene tolto dalla radice dello heap, scambiandolo con l'elemento che si trova nella penultima posizione dell'array, e così via, come si può vedere nella Figura 20.

La classe seguente realizza l'algoritmo heapsort.

**Figura 20**

Utilizzo di `heapsort` per l'ordinamento di un array



**File ch15/heapsort/HeapSorter.java**

```
/*
 * Questa classe applica l'algoritmo heapsort
 * per ordinare un array.
 */
public class HeapSorter
{
    private int[] a;

    /**
     * Costruisce un oggetto che ordina un array usando
     * l'algoritmo heapsort.
     * @param anArray un array di numeri interi
     */
    public HeapSorter(int[] anArray)
    {
        a = anArray;
    }

    /**
     * Ordina l'array gestito da questo oggetto ordinatore.
     */
    public void sort()
    {
        int n = a.length - 1;
        for (int i = (n - 1) / 2; i >= 0; i--)
            fixHeap(i, n);
        while (n > 0)
        {
            swap(0, n);
            n--;
            fixHeap(0, n);
        }
    }

    /**
     * Rende vera la proprietà di heap per un sottoalbero,
     * nell'ipotesi che i figli della sua radice la rispettino già.
     * @param rootIndex l'indice della radice del sottoalbero da sistemare
     * @param lastIndex l'ultimo indice valido all'interno dell'albero
     *                  che contiene il sottoalbero da sistemare
     */
    private void fixHeap(int rootIndex, int lastIndex)
    {
        // elimina la radice
        int rootValue = a[rootIndex];

        // promuove i figli finché sono maggiori della radice

        int index = rootIndex;
        boolean more = true;
        while (more)
        {
            int childIndex = getLeftChildIndex(index);
            if (childIndex <= lastIndex)
            {
```

```

        // usa invece il figlio destro se è maggiore
        int rightChildIndex = getRightChildIndex(index);
        if (rightChildIndex <= lastIndex
            && a[rightChildIndex] > a[childIndex])
        {
            childIndex = rightChildIndex;
        }

        if (a[childIndex] > rootValue)
        {
            // promuove il figlio
            a[index] = a[childIndex];
            index = childIndex;
        }
        else
        {
            // il valore della radice è maggiore di quello di entrambi i figli
            more = false;
        }
    }
    else
    {
        // non ci sono figli
        more = false;
    }
}

// memorizza nel nodo vuoto il valore della radice
a[index] = rootValue;
}

/**
 * Scambia due valori nell'array.
 * @param i la prima posizione da scambiare
 * @param j la seconda posizione da scambiare
 */
private void swap(int i, int j)
{
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}

/**
 * Restituisce l'indice del figlio sinistro.
 * @param index l'indice di un nodo dello heap
 * @return l'indice del figlio sinistro del nodo indicato
 */
private static int getLeftChildIndex(int index)
{
    return 2 * index + 1;
}

/**
 * Restituisce l'indice del figlio destro.
 * @param index l'indice di un nodo dello heap
 * @return l'indice del figlio destro del nodo indicato
 */

```

```

*/
private static int getRightChildIndex(int index)
{
    return 2 * index + 2;
}
}

```



## Note di cronaca 15.1

### Pirateria del software

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. Scrivere un vero prodotto software, un'applicazione amministrativa, per esempio, o un gioco per computer, richiede una gran quantità di tempo e di denaro. Poche persone, e ancor meno aziende, sono disposte a spendere tanto denaro e tempo 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 passeranno a versioni a pagamento più elaborate; altre società regalano il software che consente agli utenti di leggere e utilizzare i 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. La maggior parte dei governi fornisce una protezione legale, sotto forma di leggi sul diritto d'autore e di brevetti, per incoraggiare lo sviluppo

di nuovi prodotti. I paesi 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 *key disk*: dischetti floppy con una marcatura speciale di piccoli fori fatti con un laser, che non si potevano copiare. Altri utilizzavano *chiavi hardware* ("dongle"), da collegare alla porta della stampante del computer. Gli utenti legittimi odiavano queste misure: avevano pagato il loro software, ma dovevano sopportare il fastidio di inserire un disco chiave tutte le volte che avviavano il software o rassegnarsi ad avere parecchie chiavi hardware che penzolavano dal retro dei loro computer. Negli Stati Uniti le pressioni del mercato hanno costretto i fornitori ad abbandonare questi meccanismi di protezione contro le copie, ma sono ancora molto diffusi in altre parti del mondo.

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 pacchetto che vorreste tanto avere è troppo caro per i vostri mezzi, che 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 programmati, senza creare problemi ai clienti onesti, è un problema tuttora irrisolto e molti scienziati dell'informazione sono coinvolti in ricerche in questo campo.



## Auto-valutazione

17. Tra heapsort e mergesort, quale algoritmo richiede meno spazio di memoria?
18. Perché nella classe `HeapSorter` i calcoli dell'indice del figlio sinistro e del figlio destro vengono eseguiti in modo diverso da quanto avviene nella classe `MinHeap`?

## Riepilogo degli obiettivi di apprendimento

### Il tipo di dato astratto “insieme” e le sue implementazioni nella libreria standard

- Un insieme è una raccolta non ordinata di elementi distinti. Gli elementi vi possono essere aggiunti, rimossi e cercati.
- Gli insiemi non hanno elementi duplicati. L'aggiunta di un elemento già presente viene ignorata silenziosamente.
- Le classi `HashSet` e `TreeSet` realizzano l'interfaccia `Set`.
- Per elencare tutti gli elementi di un insieme si usa un iteratore.
- Un iteratore visita gli elementi di un `HashSet` in ordine apparentemente casuale, mentre visita ordinatamente gli elementi di un `TreeSet`.
- Non si può aggiungere un elemento in una particolare posizione di un iteratore di un insieme.

### Il tipo di dato astratto “mappa” e le sue implementazioni nella libreria standard

- Una mappa memorizza associazioni tra oggetti che fungono da chiave e oggetti che rappresentano un valore.
- Le classi `HashMap` e `TreeMap` realizzano l'interfaccia `Map`.
- Per trovare tutte le chiavi e i valori presenti in una mappa, si itera nell'insieme delle chiavi e si cerca il valore corrispondente a ciascuna chiave.

### Implementazione di una tabella hash e sue prestazioni

- Una funzione di hash calcola un valore intero a partire da un oggetto.
- Una buona funzione di hash minimizza le *collisioni*, che avvengono quando a oggetti diversi vengono associati codici di hash identici.
- Una tabella hash può essere realizzata come un array di *bucket*, sequenze di nodi che contengono elementi aventi lo stesso codice di hash.
- Se non ci sono collisioni o se ce ne sono poche, aggiungere, eliminare e cercare elementi nella tabella hash richiede un tempo costante,  $O(1)$ .

### Progettazione di un metodo `hashCode` appropriato

- Per realizzare il metodo `hashCode`, combinate i codici di hash delle variabili di esemplare.
- Il metodo `hashCode` deve essere compatibile con il metodo `equals`.
- Se una classe non sovrscrive né `equals` né `hashCode`, due suoi esemplari sono uguali soltanto quando sono lo stesso oggetto.
- Una mappa realizzata con una tabella hash usa il codice di hash delle sole chiavi.

### Implementazione di un albero di ricerca binario e sue prestazioni

- Un albero binario è composto di nodi, ciascuno dei quali ha al massimo due nodi figli.
- Tutti i nodi di un albero di ricerca binario soddisfano questa proprietà: i discendenti di sinistra di un nodo contengono dati di valore inferiore al dato contenuto nel nodo, mentre i discendenti di destra contengono dati di valore maggiore.

- Per inserire un valore in un albero di ricerca binario, lo confrontiamo ripetutamente con i valori contenuti nei nodi, scendendo verso sinistra o verso destra, fino a trovare un riferimento `null`.
- Eliminando da un albero di ricerca binario un nodo avente un solo figlio, tale figlio prende il posto del nodo eliminato.
- Per eliminare da un albero di ricerca binario un nodo avente due figli, lo si sostituisce con il nodo di valore minimo presente nel suo sottoalbero destro.
- In un albero bilanciato, tutti i percorsi che vanno dalla radice alle foglie hanno approssimativamente la stessa lunghezza.
- In un albero, le operazioni di ricerca, inserimento e rimozione di un elemento richiedono un tempo proporzionale all'altezza dell'albero.
- Se un albero di ricerca binario è bilanciato, le operazioni di ricerca, inserimento e rimozione di un elemento richiedono un tempo  $O(\log n)$ .

#### Visite di un albero in ordine simmetrico, anticipato e posticipato

- Per visitare tutti gli elementi di un albero, si visita la radice e, ricorsivamente, si visitano i sottoalberi. Distinguiamo tre tipi di visite: in ordine simmetrico, in ordine anticipato e in ordine posticipato.
- L'attraversamento in ordine posticipato di un albero di espressione fornisce le istruzioni necessarie per valutare l'espressione con una calcolatrice con funzionamento *a stack*.

#### Il tipo di dato astratto "coda prioritaria"

- Quando si elimina un elemento da una coda prioritaria viene rimosso l'elemento con la priorità più elevata.

#### Implementazione di uno **heap** e sue prestazioni

- Uno heap è un albero quasi completo in cui ogni nodo ha un valore non superiore a quelli dei propri discendenti.
- In uno heap, l'inserimento e la rimozione di un elemento sono operazioni  $O(\log n)$ .
- La disposizione regolare dei nodi di uno heap rende possibile una loro efficiente memorizzazione in un array.

#### Algoritmo **heapsort** e sue prestazioni

- L'algoritmo **heapsort** è basato sull'inserimento di elementi in uno heap e sulla loro successiva rimozione ordinata.
- L'algoritmo **heapsort** ha prestazioni  $O(n \log n)$ .

## Classi, oggetti e metodi presentati nel capitolo

|                                            |                                               |
|--------------------------------------------|-----------------------------------------------|
| <code>java.util.Collection&lt;E&gt;</code> | <code>put</code>                              |
| <code>contains</code>                      | <code>remove</code>                           |
| <code>remove</code>                        | <code>java.util.PriorityQueue&lt;E&gt;</code> |
| <code>size</code>                          | <code>remove</code>                           |
| <code>java.util.HashMap&lt;K, V&gt;</code> | <code>java.util.Set&lt;E&gt;</code>           |
| <code>java.util.HashSet&lt;K, V&gt;</code> | <code>java.util.TreeMap&lt;K, V&gt;</code>    |
| <code>java.util.Map&lt;K, V&gt;</code>     | <code>java.util.TreeSet&lt;K, V&gt;</code>    |
| <code>get</code>                           |                                               |
| <code>keyset</code>                        |                                               |

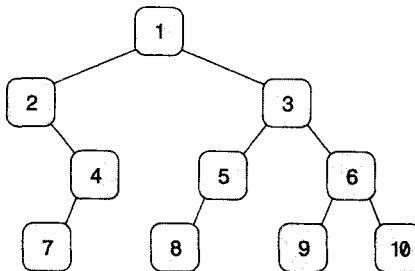
## Esercizi di ripasso

- ★ **Esercizio R15.1.** Qual è la differenza fra un insieme e una mappa?
- ★ **Esercizio R15.2.** Quali realizzazioni vengono fornite dalla libreria Java per il tipo astratto insieme?
- ★★ **Esercizio R15.3.** Quali sono le operazioni fondamentali del tipo astratto insieme? Quali metodi aggiuntivi sono forniti dall'interfaccia `Set`? Consultate la documentazione API della libreria standard.
- ★★ **Esercizio R15.4.** L'unione di due insiemi  $A$  e  $B$  è l'insieme di tutti gli elementi contenuti in  $A$ , in  $B$  o in entrambi. L'intersezione è, invece, l'insieme di tutti gli elementi contenuti sia in  $A$  sia in  $B$ . Come si possono calcolare l'unione e l'intersezione di due insiemi, usando le operazioni fondamentali per gli insiemi descritte nel Paragrafo 15.1?
- ★★ **Esercizio R15.5.** Come si possono calcolare l'unione e l'intersezione di due insiemi, usando i metodi forniti dall'interfaccia `java.util.Set`? Consultate la documentazione API della libreria standard.
- ★ **Esercizio R15.6.** Una mappa può avere due chiavi con lo stesso valore? E due valori con la stessa chiave?
- ★ **Esercizio R15.7.** Una mappa può essere realizzata mediante un insieme di coppie (*chiave, valore*). Date una spiegazione.
- ★★ **Esercizio R15.8.** Quando si realizza una mappa con tabella hash come un insieme di coppie (*chiave, valore*), come si può calcolare il codice di hash di una coppia?
- ★ **Esercizio R15.9.** Verificate i codici di hash delle stringhe "Jim" e "Joe" presentati nella Tabella 1.
- ★ **Esercizio R15.10.** Usando i codici di hash della Tabella 1, dimostrate che la Figura 5 mostra in modo accurato le posizioni delle stringhe se la dimensione della tabella hash è 101.
- ★ **Esercizio R15.11.** Qual è la differenza fra albero binario e albero di ricerca binario? Date esempi di ciascuno.
- ★ **Esercizio R15.12.** Qual è la differenza fra albero bilanciato e albero non bilanciato? Date esempi di ciascuno.
- ★ **Esercizio R15.13.** I seguenti elementi vengono inseriti in un albero di ricerca binario. Tracciate l'albero risultante dopo ciascun inserimento.

Adam  
Eve  
Romeo  
Juliet  
Tom  
Diana  
Harry

- ★★ **Esercizio R15.14.** Inserite gli elementi dell'esercizio precedente in ordine inverso. Quindi, stabilite in che modo il metodo `BinarySearchTree.print` visualizza sia l'albero dell'esercizio precedente sia questo albero. Spiegate in che modo le visualizzazioni sono fra loro correlate.

- \*\* Esercizio R15.15.** Considerate l'albero che segue. In quale ordine vengono stampati i nodi dal metodo `BinarySearchTree.print?` I nodi sono identificati dal numero, mentre i dati memorizzati non sono visibili.



- \*\* Esercizio R15.16.** Sarebbe possibile realizzare efficientemente una coda prioritaria usando un albero di ricerca binario? Fornite dettagliate argomentazioni per la vostra risposta.
- \*\*\* Esercizio R15.17.** Le visite in ordine anticipato, in ordine simmetrico o in ordine posticipato visualizzano uno heap come sequenza ordinata di valori? Perché o perché no?
- \*\*\* Esercizio R15.18.** Dimostrate che uno heap di altezza  $h$  contiene almeno  $2^{h-1}$  elementi ma meno di  $2^h$  elementi.
- \*\*\* Esercizio R15.19.** Immaginate che i nodi di uno heap vengano memorizzati in un array, a partire dalla cella di indice 1. Dimostrate che i nodi figli del nodo dello heap di indice  $i$  si trovano nelle celle di indice  $2 \times i$  e  $2 \times i + 1$ , e che il nodo genitore del nodo di indice  $i$  si trova nella cella di indice  $i / 2$ .
- \*\* Esercizio R15.20.** Simulate a mano, mostrando tutti i passi, l'algoritmo heapsort applicato a questo array:

11 27 8 14 45 6 24 81 29 33

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



# A

## Risposte alle domande di auto-valutazione

- 
1. Un programma che legge i dati presenti nel CD e li invia in uscita verso gli altoparlanti e lo schermo.
  2. Un riproduttore di CD può fare una sola cosa: suonare CD musicali. Non può eseguire programmi.
  3. No, il programma esegue semplicemente le sequenze di istruzioni che sono state predisposte dai programmatori.
  4. Nella memoria secondaria, solitamente un disco rigido.
  5. L'unità centrale di elaborazione.
  6. 21 100
  7. No, l'uso di un compilatore è destinato ai programmatori, per consentire la traduzione in codice macchina di istruzioni di programmazione di alto livello.
  8. Sicurezza e portabilità.
  9. Nessuno può conoscere l'intera libreria: è troppo vasta.
  10. 

```
System.out.println("Hello,");
    System.out.println("World!");
```
  11. Sì, la linea che inizia con // è un commento, destinato a lettori umani. Il compilatore ignora i commenti.
  12. Viene visualizzato My lucky number is12. Sarebbe utile aggiungere uno spazio vuoto prima del numero.
  13. Sì, ma bisogna ricordarsi di salvare il file in modalità "testo normale" (*plain text*).
  14. Una sequenza casuale di caratteri, alcuni dei quali piuttosto strani: i file di classe contengono istruzioni per la macchina virtuale, codificate sotto forma di numeri binari.
  15. Un errore di sintassi: il compilatore non sa come gestire la parola **visualizza**.
  16. Un errore logico: non dimenticate che il programma, per poter essere eseguito, è stato compilato.
  17. Quando un programma contiene errori di sintassi non viene generato alcun file di classe, per cui non c'è nulla da poter eseguire.
  18. Quattro anni:
    - 0 10000
    - 1 12000
    - 2 14400
    - 3 17280
    - 4 20736
  19. Il numero di minuti è inferiore a 300?  
Se sì, la risposta è  $\$ 29.95 \times 1.125 = \$ 33.70$ .

Altrimenti:

1. Calcolare la differenza “numero di minuti” – 300.
2. Moltiplicare tale differenza per 0.45.
3. Aggiungere \$29.95.
4. Moltiplicare il totale ottenuto per 1.125, trovando così la risposta.

## Capitolo 2

1. `int` e `String`
2. `double`
3. Perché un valore di tipo `int` non è un oggetto e con esso non si possono invocare metodi.
4. `(x + y) * 0.5`
5. Soltanto i primi due sono identificatori validi.
6. `String myName = "John Q. Public";`
7. No, a sinistra dell’operatore `=` deve esserci una variabile.
8. `greeting = "Hello, Nina!";`  
Notate che  
`String greeting = "Hello, Nina!";`  
non è la risposta corretta, perché in questo caso verrebbe dichiarata una nuova variabile.
9. `river.length()` oppure `"Mississippi".length()`
10. `System.out.println(greeting.toUpperCase());`  
oppure  
`System.out.println("Hello, World!".toUpperCase());`
11. No, perché la variabile `river` è di tipo `String` e in tale classe il metodo `println` non esiste.
12. Il parametro implicito è `river`; non c’è alcun parametro esplicito; il valore restituito è 11.
13. `"Missississi"`
14. 12
15. Come `public String toUpperCase()`, quindi senza parametri espliciti e con valore restituito di tipo `String`.
16. `new Rectangle(90, 90, 20, 20)`
17. 0
18. Un metodo d’accesso: non modifica la stringa originaria, ma restituisce una nuova stringa con lettere maiuscole.
19. `box.translate(-5, -10)`, a patto che il metodo venga invocato subito dopo aver memorizzato il nuovo rettangolo in `box`.
20. `toLowerCase`
21. `"Hello, Space !";` vengono eliminati soltanto gli spazi iniziali e finali.
22. Aggiungere all’inizio del programma l’enunciato  
`import java.util.Random;`

23. `x: 30, y: 25.`
24. Perché il metodo `translate` non modifica le dimensioni del rettangolo.
25. Ora `greeting` e `greeting2` fanno riferimento al medesimo oggetto di tipo `String`.
26. Entrambe le variabili fanno ancora riferimento alla medesima stringa, che non è stata modificata. Ricordate che il metodo `toUpperCase` costruisce una nuova stringa che contiene caratteri maiuscoli, senza apportare alcuna modifica alla stringa originaria.
27. Modificate il programma `EmptyFrameViewer` nel modo seguente:  
`frame.setSize(300, 300);  
frame.setTitle("Hello, World!");`
28. Costruite due oggetti di tipo `JFrame`, impostando le dimensioni di entrambi e invocando `setVisible(true)` per ciascuno di essi.
29. `Rectangle box = new Rectangle(5, 10, 20, 20);`
30. Sostituite l’invocazione `box.translate(15, 25)` con  
`box = new Rectangle(20, 35, 20, 20);`
31. Il compilatore segnala che `g` non ha un metodo `draw`.
32. `g2.draw(new Ellipse2D.Double(75, 75, 50, 50));`
33. `Line2D.Double segment1 =  
new Line2D.Double(0, 0, 10, 30);  
g2.draw(segment1);  
Line2D.Double segment2 =  
new Line2D.Double(10, 30, 20, 0);  
g2.draw(segment2);`
34. `g2.drawString("V", 0, 30);`
35. 0, 0, 255
36. Dapprima si disegna un quadrato grande, internamente colorato di rosso, poi si disegna al suo interno un quadrato piccolo, internamente colorato di giallo, in questo modo:  
`g2.setColor(Color.RED);  
g2.fill(new Rectangle(0, 0, 200, 200));  
g2.setColor(Color.YELLOW);  
g2.fill(new Rectangle(50, 50, 100, 100));`

## Capitolo 3

1. `public void reset()  
{  
 value = 0;  
}`
2. Soltanto invocando metodi della classe `Clock`.
3. In uno dei metodi della classe `Counter`.
4. I programmati che hanno progettato e realizzato la libreria di Java.
5. Altri programmati che lavorano a un’applicazione per elaborazioni finanziarie personali.

6. `harrysChecking.withdraw(harrysChecking.getBalance())`
7. Il tipo restituito dal metodo `withdraw` è `void`, per cui non restituisce alcun valore: per ottenere il saldo da visualizzare dopo il prelievo, usate il metodo `getBalance`.
8. Aggiungendo un parametro `accountNumber` ai costruttori e aggiungendo un metodo `getAccountNumber`. Non c'è bisogno di un metodo `setAccountNumber`, dato che il numero di conto non viene mai modificato dopo la creazione del conto.
9. 

```
/***
 * Questa classe costituisce un modello per
 * un contapersone.
 */
public class Counter
{
    private int value;

    /**
     * Restituisce il valore attuale del conteggio.
     * @return il valore attuale del conteggio
     */
    public int getValue()
    {
        return value;
    }

    /**
     * Incrementa di uno il valore del conteggio.
     */
    public void count()
    {
        value = value + 1;
    }
}
10. /**
 * Costruisce un nuovo conto bancario con un saldo
 * iniziale assegnato.
 * @param accountNumber il numero di conto di
 * questo conto bancario
 * @param initialBalance il saldo iniziale di
 * questo conto bancario
 */
11. Perché la prima frase della descrizione del metodo
deve, appunto, descrivere il metodo, in quanto verrà
visualizzata da sola nella tabella riassuntiva.
12. Bisogna aggiungere alla classe una variabile di esemplice:
private int accountNumber;
13. Perché si accede alla variabile di esemplare balance dal
metodo main di BankRobber. Il compilatore segnalerà
un errore, perché non si tratta di un metodo della
classe BankAccount.
14. public int getWidth()
```

- ```
{           return width;
}
```
15. Si può fare in vari modi, ad esempio:

```
public void translate(int dx, int dy)
{
    int newx = x + dx;
    x = newx;
    int newy = y + dy;
    y = newy;
}
```
16. Un oggetto di tipo `BankAccount` e nessun oggetto di tipo `BankAccountTester`. La classe `BankAccountTester` serve soltanto a contenere il metodo `main`.
17. In questo tipo di ambienti è possibile utilizzare comandi interattivi per costruire oggetti di tipo `BankAccount`, per invocarne metodi e per visualizzare i valori da essi restituiti.
18. Le variabili di entrambe le categorie appartengono a metodi: vengono create quando viene invocato il metodo e cessano di esistere quando il metodo termina la propria esecuzione. Differiscono per le modalità di inizializzazione: le variabili parametruo vengono inizializzate con i valori usati nell'invocazione del metodo, mentre le variabili locali devono essere inizializzate esplicitamente.
19. Dopo aver calcolato il resto dovuto, le variabili `payment` e `purchase` assumono il valore zero: se il metodo restituisse `payment - purchase`, restituirebbe sempre zero.
20. Un parametro implicito, `this`, di tipo `BankAccount`, e un parametro esplicito, `amount`, di tipo `double`.
21. Non è un'espressione lecita, perché `this` è di tipo `BankAccount` e la classe `BankAccount` non ha una variabile di esemplare di nome `amount`.
22. Nessun parametro implicito (il metodo `main` non viene invocato con un oggetto) e un solo parametro esplicito, di nome `args`.
23. `CarComponent`
24. Nel metodo `draw` della classe `Car`, si invoca

```
g2.fill(frontTire);
g2.fill(rearTire);
```
25. Si raddoppiano tutte le misure presenti nel metodo `draw` della classe `Car`.

## Capitolo 4

1. `int` e `double`.
2. Il Paese del mondo più popoloso è la Cina, che ha circa  $1.2 \times 10^9$  abitanti, per cui una variabile di tipo `int` può memorizzare le popolazioni di singoli Paesi. La popolazione mondiale, però, è superiore a  $6 \times 10^9$ , per

- cui, sommando popolazioni di più Paesi e calcolando popolazioni medie si può eccedere il valore massimo memorizzabile in una variabile di tipo `int`. La scelta migliore è, quindi, rappresentata dal tipo `double`. Potreste anche usare il tipo `long`, ma non si trarrebbe alcun beneficio, perché il valore della popolazione di un Paese in un determinato momento non è mai noto con precisione.
3. La prima inizializzazione non è corretta, perché il valore di destra è di tipo `double`, che non può essere assegnato a una variabile di tipo `int`. La seconda inizializzazione è, invece, corretta: un valore di tipo `int` può sempre essere convertito in un valore di tipo `double`.
  4. La prima dichiarazione è usata all'interno di un metodo, mentre la seconda all'interno di una classe.
  5. (1) Si dovrebbe usare una costante, non il "numero magico" 3.14.  
 (2) 3.14 non è un'approssimazione abbastanza precisa di  $\pi$ .
  6. Un'unità in meno del suo valore iniziale.
  7. 17 e 29.
  8. Soltanto `s3` viene diviso per 3. Per ottenere il risultato corretto bisogna usare le parentesi; inoltre, se `s1`, `s2` e `s3` sono di tipo intero, bisogna dividere per `3.0`, in modo da evitare divisioni tra interi:  

$$(s1 + s2 + s3) / 3.0$$
  9.  $\sqrt{x^2 + y^2}$
  10. Quando la parte frazionaria di  $x$  è maggiore o uguale a 0.5.
  11. Usando un cast: `(int) Math.round(x)`.
  12.  $x$  è un numero, non un oggetto: non si possono invocare metodi con numeri.
  13. No, il metodo `println` è invocato usando l'oggetto `System.out`.
  14. `s` assume il valore "Agent5".
  15. Le stringhe "i" e "ssissi".
  16. La classe ha soltanto un metodo che legge singoli byte: sarebbe molto noioso comporre caratteri, stringhe e numeri a partire da tali byte.
  17. Il valore è "John". Il metodo `next` legge la parola successiva.

## Capitolo 5

1. Se l'importo del prelievo è uguale al saldo, il saldo si deve azzerare senza alcuna penalità.
2. Soltanto il primo enunciato di assegnazione fa parte dell'enunciato `if`. Occorre usare le parentesi graffe per

raggruppare entrambi gli enunciati di assegnazione all'interno di un blocco di enunciati.

3. (a) 0; (b) 1; (c) viene lanciata un'eccezione.
4. Con errori di sintassi: e, g, h. Di dubbia utilità dal punto di vista logico: a, d, f.
5. Sì, se si invertono anche gli operatori dei confronti:  

```
if (richter < 3.5)
    r = "Generally not felt by people";
else if (richter < 4.5)
    r = "Felt by many people, no destruction";
else if (richter < 6.0)
    r = "Damage to poorly constructed buildings";
...
6. L'aliquota più elevata viene applicata al solo reddito che ricade nello scaglione più elevato. Supponete di non essere coniugati e di avere un reddito di $ 31 900. Ha senso chiedere un aumento di $ 200? Certamente sì: avrete un aumento, al netto delle tasse, pari al 90% dei primi $ 100. sommato al 75% dei rimanenti $ 100.
```
7. Quando  $x$  vale zero.
8. `if (!Character.isDigit(ch)) ...`
9. Sette.
10. Il valore 0 fornito in ingresso deve provocare la visualizzazione di "Generally not felt by people" (se viene, invece, visualizzato "Negative numbers are not allowed", c'è un errore nel programma).

## Capitolo 6

1. Non viene mai eseguito.
2. Il metodo `waitForBalance` non terminerebbe mai la propria esecuzione, per la presenza di un ciclo infinito.
3. 

```
int i = 1;
while (i <= numberOfYears)
{
    double interest = balance * rate / 100;
    balance = balance + interest;
    i++;
}
```
4. Undici volte.
5. 

```
double total = 0;
while (in.hasNextDouble())
{
    double input = in.nextDouble();
    if (value > 0) total = total + input;
}
```
6. L'invocazione iniziale di `in.nextDouble()` fallisce, provocando la terminazione del programma. Una possibile soluzione consiste nella lettura di tutti i valori all'interno del ciclo, introducendo una variabile

- booleana che tenga traccia del fatto che siamo nella prima iterazione del ciclo oppure no.
- ```
double input = 0;
boolean first = true;
while (in.hasNextDouble())
{
    double previous = input;
    input = in.nextDouble();
    if (first) { first = false; }
    else if (input == previous)
        { System.out.println("Duplicate input"); }
}
```
7. Perché non si sa se il successivo dato in ingresso sarà un numero oppure la lettera Q.
8. No. Se *tutti* i valori in ingresso fossero negativi, anche il loro massimo lo sarebbe, ma la variabile `maximum` è stata inizializzata a zero: con tale semplificazione si calcolerebbe, erroneamente, zero come valore massimo.
9. Si modifica il ciclo interno, in questo modo:
- ```
for (int j = 1; j <= width; j++)
```
10. 20.
11. int n = generator.nextInt(2);
// 0 = testa, 1 = croce
12. Perché il programma invoca ripetutamente `Math.toRadians(angle)`, mentre per calcolare  $\pi$  basta invocare semplicemente `Math.toRadians(180)`.
13. Sicuramente uno “step over”, perché non vi interessa verificare il funzionamento interno del metodo `println`.
14. Occorre impostare un breakpoint: eseguire i cicli passo dopo passo può essere assai tedioso.

## Capitolo 7

- 1, 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, ma *non* 100.
2. a. 0;
  - b. un errore in fase di esecuzione: indice di array fuori dai limiti;
  - c. un errore in fase di compilazione: c non è inizializzato.
3. new String[10];
 new ArrayList<String>();
4. `names` contiene le stringhe "B" e "C" nelle posizioni 0 e 1.
5. `double` è uno degli otto tipi primitivi, mentre `Double` è una classe.
6. `values.set(0, values.get(0) + 1);`
7. 

```
for (double element : values)
    System.out.println(element);
```
8. Conta quanti conti hanno saldo uguale a zero.

9. 

```
for (int i = valuesSize - 1; i >= 0; i--)
    System.out.println(values[i]);
valuesSize--;
```
10. Perché in un vettore di tipo `ArrayList<Double>` dovete usare oggetti involucro, che sono meno efficienti.
11. Restituisce la prima corrispondenza che trova.
12. Si, ma il primo confronto fallirebbe sempre.
13. 

```
for (int i = 0; i < names.size(); i++)
{
    System.out.print(names.get(i));
    if (i < names.size() - 1)
    {
        System.out.print(" | ");
    }
}
```
14. Se `names` fosse vuoto, nella prima riga si avrebbe un errore di limiti.
15. Perché modificando il codice è possibile introdurre errori.
16. Aggiungere al “pacchetto di prove” un caso di prova che verifichi la correzione dell’errore.
17. Perché non c’è alcun utente umano che possa vedere la richiesta, dal momento che i dati in ingresso vengono forniti in un file.
18. 

```
int[][] array = new int[4][4];
```
19. 

```
int count = 0;
for (int i = 0; i < ROWS; i++)
    for (int j = 0; j < COLUMNS; j++)
        if (board[i][j].equals(" ")) count++;
```

## Capitolo 8

1. Cercare i sostantivi nella descrizione del problema.
2. Sì (`ChessBoard`) e no (`MovePiece`).
3. Alcune delle sue caratteristiche riguardano i pagamenti, altre i valori delle monete.
4. Nessuna delle sue operazioni necessita della classe `CashRegister`.
5. Se una classe non dipende da un’altra, non risulta affetta da modifiche apportate a quest’ultima.
6. È un metodo d’accesso: l’invocazione di `substring` non modifica la stringa con cui il metodo viene invocato. A dire il vero, tutti i metodi della classe `String` sono metodi d’accesso.
7. No, `translate` è un suo metodo modificatore.
8. È un effetto collaterale, un tipo di effetto collaterale molto comune nella programmazione orientata agli oggetti.
9. Sì, perché il metodo modifica lo stato del suo parametro di tipo `Scanner`.

10. In tal modo non vi dovete preoccupare di verificare che i valori su cui il metodo opera siano validi: tale azione diventa una responsabilità di chi invoca il metodo.
11. No, potete intraprendere qualunque azione riteniate opportuna.
12. `Math m = new Math(); y = m.sqrt(x);`
13. Non potreste aggiungere un metodo di esemplare alla classe `ArrayList`: è una classe della libreria standard, che non potete modificare.
14. `System.in` e `System.out`.
15. Sì, funziona, perché i metodi statici possono accedere alle variabili statiche della classe a cui appartengono, ma è una strategia pessima: al complicarsi degli obiettivi del vostro progetto, avrete bisogno di usare oggetti e classi, per organizzare meglio il vostro programma.
16. Sì, i due ambiti di visibilità sono disgiunti.
17. Si estende dall'inizio alla fine della classe.
18. (a) No; (b) Sì; (c) Sì; (d) No.
19. No, basta usare per tutte le altre classi nomi completi, come `java.util.Random` e `java.awt.Rectangle`.
20. `/home/me/cs101/hw1/problem1` oppure, in Windows, `c:\Users\Me\cs101\hw1\problem1`.
21. Ecco una possibile risposta:

```
public class EarthquakeTest {
    @Test public void testLevel4() {
        Earthquake quake = new Earthquake(4);
        Assert.assertEquals(
            "Felt by many people, no destruction",
            quake.getDescription());
    }
}
```
22. È una soglia di tolleranza per il confronto di numeri in virgola mobile: vogliamo che la prova sia considerata superata anche se si presenta un piccolo errore di arrotondamento.

## Capitolo 9

1. Deve realizzare l'interfaccia `Measurable` e il suo metodo `getMeasure` deve fornire la popolazione.
2. La classe `Object` non ha un metodo `getMeasure`, mentre il metodo `add` deve invocare proprio tale metodo.
3. Soltanto se `meas` si riferisce veramente a un oggetto di tipo `BankAccount`.
4. No, un riferimento di tipo `Coin` può essere convertito in un riferimento di tipo `Measurable`, ma se cercate

- di convertirlo, con un cast, in un riferimento di tipo `BankAccount`, viene lanciata un'eccezione.
5. Perché `Measurable` è un'interfaccia e le interfacce non hanno variabili di esemplare, né codice che ne realizzzi i metodi.
6. Una tale variabile non fa mai riferimento a un oggetto di tipo `Measurable`, ma a un oggetto di una classe che realizzzi l'interfaccia `Measurable`.
7. Il frammento di codice visualizza 500.05. Ogni invocazione di `add` provoca l'invocazione di `x.getMeasure()`: nella prima invocazione, `x` fa riferimento a un oggetto di tipo `BankAccount`, mentre nella seconda fa riferimento a un oggetto di tipo `Coin`. Nei due casi, quindi, vengono invocati due diversi metodi `getMeasure`, il primo dei quali restituisce il saldo di un conto, mentre il secondo restituisce il valore di una moneta.
8. La classe `String` non realizza l'interfaccia `Measurable`.
9. Progettate una classe `StringMeasurer` che realizzi l'interfaccia `Measurer`.
10. Un “misuratore” misura un oggetto, mentre il metodo `getMeasure` misura “se stesso”, cioè il proprio parametro implicito.
11. Le classi interne sono molto utili per realizzare classi che non rappresentano un concetto particolarmente significativo. Inoltre, i loro metodi hanno accesso alle variabili locali e di esemplare dell’ambito di visibilità circostante.
12. Quattro: uno per la classe esterna, uno per la classe interna, uno per la classe `DataSet` e uno per l’interfaccia `Measurer`.
13. Perché volete realizzare la classe `GradingProgram` usando quell’interfaccia, in modo da non dover fare modifiche quando non userete più la classe `mock`, sostituendola con una classe completamente funzionante.
14. Perché il progettista della classe `GradingProgram` non deve aspettare che la classe `GradeBook` venga compilata.
15. L’oggetto `button` è la sorgente di eventi, mentre l’oggetto `listener` è il ricevitore.
16. Perché la classe `ClickListener` realizza l’interfaccia `ActionListener`.
17. L’accesso diretto è più semplice di quella che ne sarebbe un’alternativa: passare la variabile come parametro di un costruttore o di un metodo.
18. La variabile locale deve essere dichiarata `final`.
19. Bisogna aggiungere al pannello prima `label`, poi `button`.

20. Perché il metodo `actionPerformed` non accede a tale variabile.
21. Il temporizzatore ha la necessità di invocare un metodo ogni volta che scade il proprio intervallo di temporizzazione, per cui invoca il metodo `actionPerformed` dell'oggetto ricevitore di eventi.
22. Il rettangolo verrebbe disegnato nella sua nuova posizione soltanto quando il componente viene ridisegnato per qualche motivo indipendente dal programma, come, ad esempio, quando la sua finestra viene ridimensionata dall'utente.
23. Perché si conosce la posizione attuale del mouse, non l'entità del suo spostamento.
24. Perché realizza l'interfaccia `MouseListener`, che ha cinque metodi.

## Capitolo 10

1. A rappresentare il comportamento comune a campi di testo e a componenti di testo.
2. Perché in alcuni conti bancari non maturano interessi.
3. Due variabili di esemplare: `balance` e `interestRate`.
4. `deposit`, `withdraw`, `getBalance` e `addInterest`.
5. `Manager` è la sottoclasse, `Employee` è la superclasse.
6. La classe `SavingsAccount` eredita i metodi `deposit`, `withdraw` e `getBalance`, mentre il metodo `addInterest` è nuovo. Nessun metodo sovrascrive un metodo della superclasse.
7. Deve diminuire il saldo ma non può accedere direttamente alla variabile di esemplare `balance`.
8. Perché così tale contatore è in grado di funzionare correttamente nel mese seguente.
9. Perché può usare il costruttore predefinito della superclasse, che imposta a zero il saldo.
10. No, quello è un requisito dei soli costruttori. Ad esempio, il metodo `CheckingAccount.deposit` incrementa per prima cosa il contatore di transazioni, poi invoca il metodo della superclasse.
11. Perché vogliamo usare quel metodo con qualsiasi tipo di conto bancario. Se usassimo un parametro di tipo `SavingsAccount`, non potremmo invocare il metodo con un oggetto di tipo `CheckingAccount`.
12. Perché non possiamo invocare il metodo `deposit` con una variabile di tipo `Object`.
13. L'oggetto è un esemplare di `BankAccount` o di una sua sottoclasse.
14. Il saldo di `a` non viene modificato (le operazioni di prelievo e versamento riguardano il medesimo conto),

mentre il contatore delle transazioni viene incrementato due volte.

15. Certamente sì, a meno che, ovviamente, `x` non valga `null`.
16. Se `toString` restituisce una stringa che descrive tutte le variabili di esemplare, potete semplicemente invocare `toString` con il parametro implicito ed esplicito, confrontando le stringhe ottenute. Il confronto delle variabili di esemplare, però, è più efficiente della loro conversione in stringhe.
17. Tre: `InvestmentFrameViewer`, `InvestmentFrame` e `BankAccount`.
18. Il costruttore di `InvestmentFrame` aggiunge il pannello *a se stesso*.

## Capitolo 11

1. Quando viene creato l'oggetto di tipo `PrintWriter`, il file di uscita viene reso vuoto: purtroppo, si trattava anche del file di ingresso, che risulta ora vuoto, e il ciclo `while`, di conseguenza, termina immediatamente.
2. Il costruttore di `Scanner` lancia un'eccezione di tipo `FileNotFoundException` e il programma termina la propria esecuzione.
3. `number` vale 6, `input` è uguale a `"995.0"`.
4. `price` assume il valore 6, perché, in Java, la virgola non viene considerata parte di un numero espresso in virgola mobile. Successivamente, l'invocazione di `nextInt` provoca il lancio di un'eccezione e non viene assegnato alcun valore a `quantity`.
5. Leggendoli come stringhe e convertendo in numeri quelle che sono diverse da N/A:

```
String input = in.next();
if (!input.equals("N/A"))
{
    double value = Double.parseDouble(input);
    Elabora value
}
```
6. Lanciando un'eccezione se la quantità di denaro versata è negativa.
7. Il saldo è ancora zero perché l'ultimo enunciato del metodo `withdraw` non è mai stato eseguito.
8. Dovete necessariamente includere `FileNotFoundException` e potete aggiungere anche `NoSuchElementException`, se la cosa vi sembra utile a fini di documentazione del codice. L'eccezione `InputMismatchException` è una sottoclasse di `NoSuchElementException`, quindi spetta a voi decidere se aggiungerla o meno.
9. Perché i programmati dovrebbero semplicemente verificare che i puntatori non abbiano il valore `null`

- prima di utilizzarli, anziché cercare di gestire un'eccezione di tipo `NullPointerException`.
10. Il costruttore di `Scanner` agisce con successo e l'oggetto `in` viene costruito. Successivamente, l'invocazione `in.next()` lancia un'eccezione di tipo `NoSuchElementException` e l'esecuzione del blocco `try` termina prematuremente. Nessuna delle clausole `catch` corrisponde all'eccezione lanciata, per cui nessuna di esse viene eseguita. Se nessuno dei metodi presenti nella catena di invocazione cattura l'eccezione, il programma termina.
  11. No, i due tipi di eccezioni si catturano nello stesso modo.
  12. Se fosse stata dichiarata all'interno del blocco `try`, il suo ambito di visibilità si sarebbe esteso soltanto fino alla fine di esso e la clausola `finally` non avrebbe potuto invocare `close`.
  13. Il costruttore di `PrintWriter` lancia un'eccezione, per cui non viene eseguito l'assegnamento alla variabile `out` e viene ignorato il blocco `try`. La clausola `finally`, correttamente, non viene eseguita, coerentemente con la mancata inizializzazione di `out`.
  14. Passare alla superclasse `RuntimeException` la stringa contenente il messaggio descrittivo dell'eccezione.
  15. Dal momento che l'eventuale corruzione del file non è sotto il controllo del programmatore, si dovrebbe usare un'eccezione controllata, per cui sarebbe un errore estendere `RuntimeException` o `IllegalArgumentExeption`. Dato che l'errore è legato alle attività in ingresso, `IOException` sarebbe una buona scelta.
  16. Perché non potrebbe fare niente di sensato. La classe `DataSetReader` è una classe riutilizzabile che può essere impiegata in sistemi con diverse interfacce verso l'utente e, forse, anche con lingue diverse, per cui non può assolutamente dialogare con l'utente del programma.
  17. `DataAnalyzer.main` invoca `DataSetReader.readFile`, che a sua volta invoca `readData`. L'invocazione `in.hasNextInt()` restituisce `false` e `readData` lancia un'eccezione di tipo `BadDataException`. Il metodo `readFile` non cattura tale eccezione, che viene quindi propagata fino al metodo `main`, dove viene catturata.

## Capitolo 12

1. Immaginate di eliminare tale enunciato. Quando viene calcolata l'area del triangolo di ampiezza 1, si calcola l'area del triangolo di ampiezza 0, che ha il valore 0, e si aggiunge 1, ottenendo il valore corretto dell'area.

2. Occorre calcolare ricorsivamente l'area del quadrato di dimensione inferiore, poi restituire

```
smallerArea + width + width - 1
```

```
[][][][]
[][][]
[][][]
[][][]
```

Ovviamente, sarebbe più semplice calcolare l'area come `width * width`. I risultati sarebbero identici, perché:

$$1 + 0 + 2 + 1 + 3 + 2 + \dots + n - 1 =$$

$$= \frac{n(n+1)}{2} + \frac{(n-1)n}{2} = n^2$$

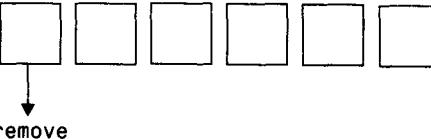
3. No, il primo metodo potrebbe avere un nome diverso, come `substringIsPalindrome`.
4. Quando `start >= end`, cioè quando la stringa esaminata è vuota oppure contiene un solo carattere.
5. Il ciclo è un po' più veloce e, ovviamente, è ancora più rapido calcolare, semplicemente, `width * (width + 1) / 2`.
6. No, la soluzione ricorsiva è quasi veloce come quella iterativa: entrambe richiedono  $n - 1$  moltiplicazioni per il calcolo di  $n!$ .
7. Sono: b seguita dalle sei permutazioni di eat, e seguita dalle sei permutazioni di bat, a seguita dalle sei permutazioni di bet e t seguita dalle sei permutazioni di bea.
8. Basta semplicemente modificare `if (word.length() == 0)` in `if (word.length() <= 1)`, perché una parola di una sola lettera è anche l'unica permutazione di se stessa.
9. Una soluzione iterativa dovrebbe avere un ciclo il cui corpo calcola la permutazione successiva a partire da quelle calcolate in precedenza, ma non esiste un algoritmo ovvio per calcolare la permutazione successiva. Ad esempio, avendo già scoperto le permutazioni eat, eta e aet, non è chiaro come usare questa informazione per ottenere la permutazione successiva. In realtà, esiste un meccanismo, piuttosto ingegnoso, per farlo, ma è ben lontano dall'essere ovvio, come si può vedere nell'Esercizio P12.12.
10. I fattori vengono combinati tra loro mediante operatori moltiplicativi (`* e /`), mentre ai termini sono applicati operatori additivi (`+ e -`): ci servono entrambi questi concetti per fare in modo che la moltiplicazione abbia la precedenza rispetto all'addizione.
11. Per gestire le operazioni con parentesi, come `2+3*(4+5)`. La sotto-espressione `4+5` è gestita da un'invocazione ricorsiva di `getExpressionValue`.

12. L'invocazione di `Integer.parseInt` in `getFactorValue` lancia un'eccezione quando trova la stringa `")"`.

## Capitolo 13

- Non si può eliminare la variabile `temp`: `a[i]` e `a[j]` verrebbero ad avere il medesimo valore.
- $1|54326, 12|4356, 123|456, 1234|56, 12345|6$ .
- Il quadruplo del tempo necessario per ordinare 40 000 valori, cioè circa 40 secondi.
- Una parabola.
- Occorre un tempo circa 100 volte superiore.
- Se  $n$  vale 4, allora  $(1/2)n^2$  vale 8 e  $(5/2)n - 3$  vale 7.
- Quando il precedente ciclo `while` termina, la condizione del ciclo deve essere falsa, per cui `iFirst >= first.length oppure iSecond >= second.length` (legge di De Morgan).
- Per prima cosa si ordina 8 7 6 5. Ricorsivamente, prima si ordina 8 7. Ricorsivamente, prima si ordina 8, che è ordinato, poi si ordina 7, che è ordinato. Si fondono i due, ottenendo 7 8. Facendo la stessa cosa con 6 5 si ottiene 5 6. Si fondono questi due risultati, ottenendo 5 6 7 8. Si fa la stessa cosa con 4 3 2 1: si ordina 4 3, ordinando 4 e 3 e fondendoli per ottenere 3 4; si ordina 2 1, ordinando 2 e 1 e fondendoli per ottenere 1 2; si fondono 3 4 e 1 2, ottenendo 1 2 3 4. Infine, si fondono 5 6 7 8 e 1 2 3 4, ottenendo 1 2 3 4 5 6 7 8.
- Circa  $(100\,000 \times \log(100\,000)) / (50\,000 \times \log(50\,000)) \approx 2 \times 5/4.7 = 2.13$  volte il tempo richiesto per ordinare 50 000 valori, cioè  $2.13 \times 97$  millisecondi, circa 207 millisecondi.
- $2n \log(2n) / (n \log(n)) = 2(1 + \log 2) / \log(n)$ . Per  $n > 2$ , si tratta di un valore minore di 3.
- In media servono 500 000 confronti.
- Il metodo `search` restituisce l'indice della posizione in cui avviene il ritrovamento, non il dato memorizzato in tale posizione.
- Circa 20 (il logaritmo in base 2 di 1024 è 10).
- In questo modo sapete dove inserire l'elemento in modo che l'array rimanga ordinato, potendo così continuare a utilizzare la ricerca binaria.
- Se il metodo restituisse  $-k$ , ottenendo il valore 0 non si saprebbe se il valore cercato è presente nell'array oppure no.
- Perché la classe `Rectangle` non realizza l'interfaccia `Comparable`.
- Bisogna che la classe `BankAccount` realizzi l'interfaccia `Comparable` e che il suo metodo `compareTo` confronti i saldi dei conti bancari.

## Capitolo 14

- Sì, per due motivi: occorre memorizzare i riferimenti ai nodi e ciascun nodo è un oggetto (per memorizzare un oggetto nella macchina virtuale c'è bisogno di una quantità fissa di memoria aggiuntiva rispetto ai dati in esso contenuti).
- Perché per accedere a qualsiasi posizione nell'array è sufficiente un numero intero usato come indice.
- Quando la lista è vuota, `first` vale `null`. Viene creato un nuovo esemplare di `Node`, alla cui variabile di esemplare `data` viene assegnato il riferimento al nuovo oggetto inserito nella lista e alla cui variabile di esemplare `next` viene assegnato il valore `null`, perché `first` vale `null`. Alla variabile di esemplare `first` viene assegnato il riferimento al nuovo nodo e, come risultato, si ha una lista concatenata di lunghezza 1.
- Punta all'elemento di sinistra. Lo si può vedere analizzando la prima invocazione di `next`, al termine della quale `position` punta al primo nodo.
- Se `position` vale `null`, significa che si è all'inizio della lista e l'inserimento di un elemento richiede l'aggiornamento del riferimento `first`. Se ci si trova in una posizione intermedia all'interno della lista, il riferimento `first` non deve essere modificato.
- Ci si può concentrare sulle caratteristiche fondamentali del tipo di dati senza essere distratti da dettagli realizzativi.
- La visione astratta è simile a quella della Figura 9, con frecce in entrambe le direzioni. La visione concreta è simile a quella della Figura 8, con l'aggiunta, in ciascun nodo, del riferimento al nodo precedente.
- Per localizzare l'elemento centrale servono  $n/2$  passi. Per localizzare l'elemento centrale della porzione di sinistra o di destra servono, poi, altri  $n/4$  passi e la ricerca successiva richiede  $n/8$  passi. Di conseguenza, per localizzare un elemento servono, nel caso peggiore,  $n$  passi. A questo punto, è meglio fare una semplice ricerca lineare, che richiede, in media,  $n/2$  passi.
- 
- La pila usa la strategia "l'ultimo a entrare è il primo a uscire". Se inviate una stampa per primi e, poi, molte altre persone stampano prima che la stampante abbia avuto modo di gestire la vostra richiesta, queste ultime persone vedranno soddisfatte le loro richieste prima

di voi: dovrete attendere che le loro stampe terminino prima di veder uscire la vostra.

## Capitolo 15

1. Un insieme realizzato in modo efficiente è in grado di verificare velocemente se un determinato elemento sia presente nell'insieme stesso.
2. Gli insiemi non hanno una proprietà di ordinamento posizionale, per cui non ha senso aggiungere un elemento in corrispondenza di una specifica posizione assunta da un iteratore, né ha senso scandire a ritroso un insieme.
3. Le parole saranno elencate in ordine.
4. Quando è preferibile visitare gli elementi dell'insieme secondo il proprio ordinamento caratteristico.
5. Un insieme memorizza elementi, mentre una mappa memorizza associazioni tra chiavi e valori.
6. L'ordinamento delle chiavi è ininfluente e non possono esserci chiavi duplicate.
7. Sì, l'insieme realizzato con tabella hash funzionerebbe correttamente: tutti gli elementi verrebbero inseriti in un unico bucket.
8. Identifica il successivo bucket all'interno dell'array di bucket e fa riferimento al suo primo elemento.
9.  $31 \times 116 + 111 = 3707$ .

10. 13.
11. In un albero, ogni nodo può avere un numero di figli qualsiasi. In un albero binario, ogni nodo può avere al massimo due figli. In un albero binario bilanciato, ogni nodo deve avere, approssimativamente, tanti discendenti a sinistra quanti ne ha a destra.
12. Ad esempio, Sarah, così come qualsiasi stringa compresa tra Romeo e Tom.
13. Per entrambi gli alberi l'attraversamento in ordine simmetrico è: 3 + 4 \* 5.
14. No. Considerate, ad esempio, i figli del nodo +: anche senza cercare i codici Unicode corrispondenti a 3, 4 e +, è evidente che + non è compreso tra 3 e 4.
15. Una coda prioritaria, perché vogliamo estrarre prima gli eventi più importanti, anche se sono avvenuti più tardi di altri eventi.
16. Sì, ma un albero di ricerca binario non è quasi pieno, per cui l'array potrebbe avere zone "vuote". I nodi mancanti potrebbero essere rappresentati da celle dell'array contenenti il valore null.
17. Heapsort richiede meno spazio di memoria, perché non necessita di array ausiliari.
18. La classe MinHeap spreca la cella di indice 0 per rendere più intuitive le formule. Quando si ordina un array, non vogliamo ignorare la cella 0, per cui modifichiamo le formule.

# 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
<code>.</code>	Accesso alle caratteristiche di una classe	da sinistra a destra
<code>[]</code>	Indice di array	da sinistra a destra
<code>()</code>	Invocazione di metodo	da sinistra a destra
<code>++</code>	Incremento	da destra a sinistra
<code>--</code>	Decremento	da destra a sinistra
<code>!</code>	Not booleano	da destra a sinistra
<code>-</code>	Not bit a bit	da destra a sinistra
<code>+ (unario)</code>	(nessun effetto)	da destra a sinistra
<code>- (unario)</code>	Inverte il segno	da destra a sinistra
<code>(NomeTipo)</code>	Cast	da destra a sinistra
<code>new</code>	Creazione di oggetto	da destra a sinistra
<code>*</code>	Moltiplicazione	da sinistra a destra
<code>/</code>	Divisione o divisione intera	da sinistra a destra
<code>%</code>	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 sinistra a destra
=	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
<code>abstract</code>	Una classe o un metodo astratti
<code>assert</code>	Un'asserzione di una condizione che deve essere verificata
<code>boolean</code>	Il tipo booleano
<code>break</code>	Interrompe l'attuale ciclo o enunciato con etichetta
<code>byte</code>	Il tipo intero a 8 bit
<code>case</code>	Un'etichetta in un enunciato <code>switch</code>
<code>catch</code>	Il gestore di un'eccezione in un blocco <code>try</code>
<code>char</code>	Il tipo carattere Unicode a 16 bit
<code>class</code>	Definisce una classe
<code>const</code>	Non usata
<code>continue</code>	Ignora la parte restante del corpo di un ciclo
<code>default</code>	L'etichetta predefinita in un enunciato <code>switch</code>
<code>do</code>	Un ciclo il cui corpo viene eseguito almeno una volta
<code>double</code>	Il tipo in virgola mobile con doppia precisione a 64 bit
<code>else</code>	La clausola alternativa in un enunciato <code>if</code>
<code>enum</code>	Un tipo enumerativo
<code>extends</code>	Indica che una classe è una sottoclasse di un'altra
<code>final</code>	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
<code>finally</code>	Una clausola di un blocco <code>try</code> che viene sempre eseguita

(continua)

(seguito)

<b>Parola riservata</b>	<b>Descrizione</b>
<code>float</code>	Il tipo in virgola mobile con singola precisione a 32 bit
<code>for</code>	Un ciclo con inizializzazione, condizione e espressioni di aggiornamento
<code>goto</code>	Non usata
<code>if</code>	Un enunciato per una diramazione condizionata
<code>implements</code>	Indica che una classe realizza un'interfaccia
<code>import</code>	Consente l'uso di nomi di classe senza il nome del pacchetto
<code>instanceof</code>	Verifica se il tipo di un oggetto è uguale al tipo specificato o a una sua sottoclasse
<code>int</code>	Il tipo intero a 32 bit
<code>interface</code>	Un tipo astratto con soli metodi astratti e costanti
<code>long</code>	Il tipo intero a 64 bit
<code>native</code>	Un metodo realizzato in un linguaggio diverso da Java
<code>new</code>	Crea un nuovo oggetto
<code>package</code>	Una raccolta di classi correlate
<code>private</code>	Una caratteristica che è accessibile soltanto da metodi della stessa classe
<code>protected</code>	Una caratteristica che è accessibile soltanto da metodi della stessa classe, di una sottoclasse o di un'altra classe nello stesso pacchetto
<code>public</code>	Una caratteristica che è accessibile da qualsiasi metodo
<code>return</code>	Termina un metodo
<code>short</code>	Il tipo intero a 16 bit
<code>static</code>	Una caratteristica definita per una classe, invece che per singoli esemplari
<code>strictfp</code>	Usa regole stringenti per i calcoli in virgola mobile
<code>super</code>	Invoca il costruttore della superclasse o un suo metodo
<code>switch</code>	Un enunciato di selezione
<code>synchronized</code>	Un blocco di codice che è accessibile da un solo thread per volta
<code>this</code>	Il parametro implicito di un metodo, oppure l'invocazione di un altro costruttore della classe
<code>throw</code>	Lancia un'eccezione
<code>throws</code>	Le eccezioni che possono essere lanciate da un metodo
<code>transient</code>	Campi che non dovrebbero essere serializzati
<code>try</code>	Un blocco di codice con gestori di eccezione o con un gestore <code>finally</code>
<code>void</code>	Contrassegna un metodo che non restituisce alcun valore
<code>volatile</code>	Un campo che potrebbe essere aggiornato da più thread senza sincronizzazione
<code>while</code>	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 storici sono stati messi a punto da persone che contavano con le dita delle mani. 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 elettronico. 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 separatore decimale, si usano le potenze negative di 2.

$$\begin{aligned}1.101_{\text{binario}} &= 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 1 + 1/2 + 1/8 = 1 + 0.5 + 0.125 \\&= 1.625\end{aligned}$$

**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

In generale, per convertire un numero binario nel suo equivalente decimale, valutate semplicemente le potenze di 2 che corrispondono alle cifre di valore 1 e sommatele. La Tabella 1 mostra le prime potenze di 2.

Per convertire in binario un numero intero decimale, dividetelo ripetutamente per 2, tenendo traccia dei resti e fermatevi quando il dividendo diventa 0. Scrivete quindi i resti come numero binario, iniziando dall'*ultimo*. Ad esempio

$$\begin{aligned}
 100 \div 2 &= 50 \text{ resto } \mathbf{0} \\
 50 \div 2 &= 25 \text{ resto } \mathbf{0} \\
 25 \div 2 &= 12 \text{ resto } \mathbf{1} \\
 12 \div 2 &= 6 \text{ resto } \mathbf{0} \\
 6 \div 2 &= 3 \text{ resto } \mathbf{0} \\
 3 \div 2 &= 1 \text{ resto } \mathbf{1} \\
 1 \div 2 &= 0 \text{ resto } \mathbf{1}
 \end{aligned}$$

Quindi,  $100_{\text{decimale}} = 1100100_{\text{binario}}$ .

Per convertire, invece, un numero frazionario minore di 1 nel formato binario, moltiplicatelo ripetutamente per 2. Se il risultato è maggiore di 1, sottraete 1; fermatevi quando il numero da moltiplicare diventa 0. Scrivete quindi le cifre che precedono il punto decimale come cifre binarie della parte frazionaria, iniziando dalla *prima*. Ad esempio

$$\begin{aligned}
 0.35 \cdot 2 &= \mathbf{0.7} \\
 0.7 \cdot 2 &= \mathbf{1.4} \\
 0.4 \cdot 2 &= \mathbf{0.8} \\
 0.8 \cdot 2 &= \mathbf{1.6} \\
 0.6 \cdot 2 &= \mathbf{1.2} \\
 0.2 \cdot 2 &= \mathbf{0.4} \\
 &\vdots
 \end{aligned}$$

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.

## 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 negativo:

- Cambiate il valore di tutti i bit della rappresentazione binaria del suo valore assoluto.
- Quindi, aggiungete 1.

Ad esempio, per calcolare  $-13$  come valore di 8 bit, dapprima cambiate il valore di tutti i bit di 00001101, ottenendo 11110010, quindi aggiungete 1:

$$-13 = 11110011_{\text{complemento a due}}$$

Ora, 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:

$$\begin{array}{r} 1 \ 1111 \ 111 \\ +13 \quad 0000 \ 1101 \\ -13 \quad 1111 \ 0011 \\ 1 \ 0000 \ 0000 \end{array}$$

Sono importanti, però, soltanto gli 8 bit più a destra, 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 dato 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:

```
byte 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).

## Numeri in virgola mobile nello standard IEEE

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  $b_0b_1b_2b_3\dots$  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 nel sistema di numerazione binario il primo bit di una rappresentazione normalizzata deve necessariamente essere 1, in realtà 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, ma 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.

**Figura 1**  
Rappresentazione IEEE  
per numeri  
in virgola mobile

1 bit	8 bit	23 bit
Segno	Esponente con bias <i>e</i> + 127	Mantissa (senza 1 iniziale)

Singola precisione

1 bit	11 bit	52 bit
Segno	Esponente con bias <i>e</i> + 1023	Mantissa (senza 1 iniziale)

Doppia precisione

Quindi

$$100_{\text{decimale}} = 0|10000101|10010000000000000000000000000000_{\text{IEEE singola precisione}}$$

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  $\neq \pm 0$ .

## Numeri esadecimali

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 ad 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`.

**Tabella 2**  
Cifre esadecimali

	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



# E

## Operazioni con bit e scorrimenti

In Java esistono quattro operazioni che agiscono su bit: la negazione unaria ( $\sim$ ) e le operazioni binarie and ( $\&$ ), or ( $\mid$ ) e or esclusivo ( $\wedge$ ), detto anche xor.

**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 $\wedge$ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Le Tabelle 1 e 2 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 svolta sui bit corrispondenti.

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 intero a 32 bit, 000000000000000000000000101110) e  $13_{\text{decimale}} = 1101_{\text{binario}}$ . Poi, effettuate l'operazione sui bit corrispondenti:

$$\begin{array}{r}
 0 \dots 0101110 \\
 \& 0 \dots 0001101 \\
 \hline
 0 \dots 0001100
 \end{array}$$

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:

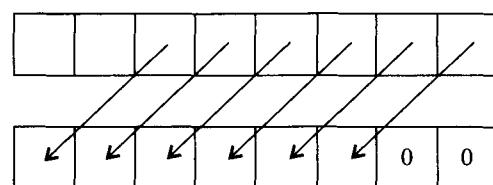
$$\begin{array}{r}
 0 \dots 0000001 \\
 | 0 \dots 0000010 \\
 \hline
 0 \dots 0000011
 \end{array}$$

Non confondete gli operatori per i bit `&` e `|` con gli operatori `&&` e `||`. Questi ultimi operano soltanto su valori di tipo `boolean`, non sui bit.

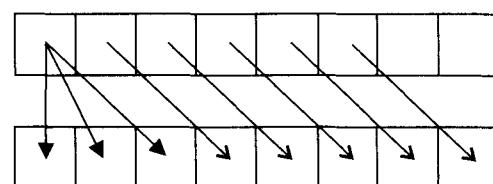
Oltre alle operazioni che operano su singoli bit, esistono anche tre operazioni di *scorrimento* che prendono lo schema di bit di un numero e lo spostano a sinistra o a destra di un certo numero di posizioni. Esistono tre operazioni di scorrimento: scorrimento a sinistra (`<<`), scorrimento aritmetico a destra (`>>`) e scorrimento a destra bit a bit (`>>>`).

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 aritmetico verso destra 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 bit a bit sposta tutti i bit a destra, inserendo zeri nei bit più significativi (osservate la Figura 1).

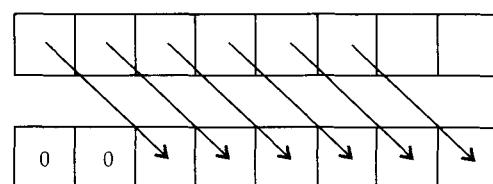
**Figura 1**  
Le operazioni  
di scorrimento



Scorrimento a sinistra (`<<`)



Scorrimento aritmetico a destra (`>>`)



Scorrimento a destra bit a bit (`>>>`)

Notate che il valore a destra dell'operatore di scorrimento viene usato modulo 32 (per valori di tipo `int`) o 64 (per valori di tipo `long`), per determinare il vero 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 << n
```

fornisce uno schema di bit in cui il bit  $n$ -esimo vale 1 (contando le posizioni a partire dalla posizione 0 del bit meno significativo).

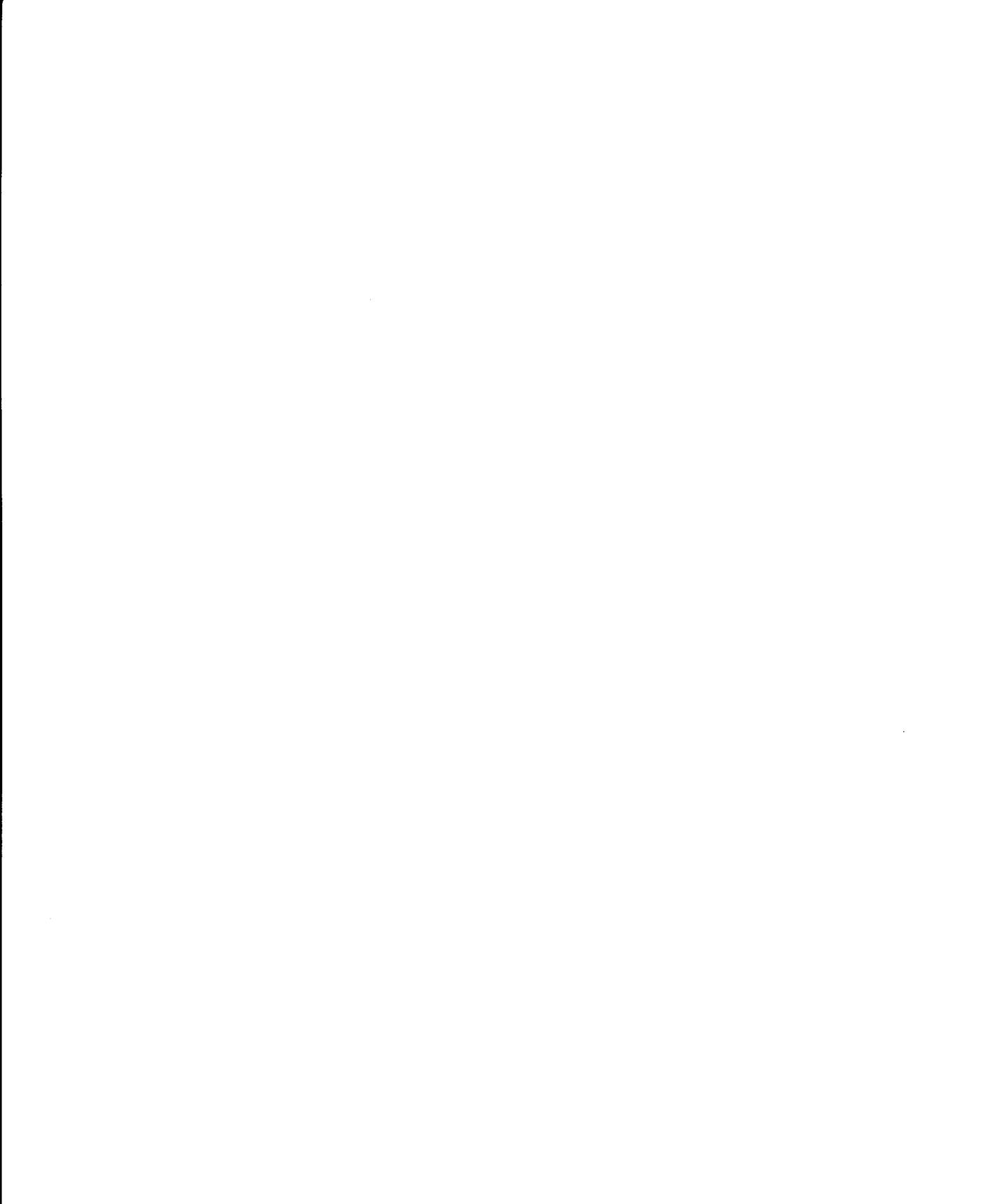
Per impostare a 1 il bit  $n$ -esimo di un numero, eseguite l'operazione

```
x = x | 1 << n
```

Per controllare se il bit  $n$ -esimo di un numero vale 1, eseguite la verifica

```
if ((x & 1 << n) != 0) ...
```

Notate che le parentesi attorno all'operatore `&` sono necessarie, perché ha una precedenza minore degli operatori relazionali.



# F

## Il sottoinsieme Basic Latin di Unicode

**Tabella 1**  
Il sottoinsieme Basic Latin  
(ASCII) di Unicode

Carattere	Codice	Decimale	Carattere	Codice	Decimale
!	'\u0021'	33	@	'\u0040'	64
"	'\u0022'	34	A	'\u0041'	65
#	'\u0023'	35	B	'\u0042'	66
\$	'\u0024'	36	C	'\u0043'	67
%	'\u0025'	37	D	'\u0044'	68
&	'\u0026'	38	E	'\u0045'	69
'	'\u0027'	39	F	'\u0046'	70
(	'\u0028'	40	G	'\u0047'	71
)	'\u0029'	41	H	'\u0048'	72
★	'\u002A'	42	I	'\u0049'	73
+	'\u002B'	43	J	'\u004A'	74
,	'\u002C'	44	K	'\u004B'	75
-	'\u002D'	45	L	'\u004C'	76
.	'\u002E'	46	M	'\u004D'	77
/	'\u002F'	47	N	'\u004E'	78
0	'\u0030'	48	O	'\u004F'	79
1	'\u0031'	49	P	'\u0050'	80
2	'\u0032'	50	Q	'\u0051'	81
3	'\u0033'	51	R	'\u0052'	82
4	'\u0034'	52	S	'\u0053'	83
5	'\u0035'	53	T	'\u0054'	84

(continua)

(seguito)

Carattere	Codice	Decimale	Carattere	Codice	Decimale
6	'\u0036'	54	U	'\u0055'	85
7	'\u0037'	55	V	'\u0056'	86
8	'\u0038'	56	W	'\u0057'	87
9	'\u0039'	57	X	'\u0058'	88
:	'\u003A'	58	Y	'\u0059'	89
;	'\u003B'	59	Z	'\u005A'	90
<	'\u003C'	60	[	'\u005B'	91
=	'\u003D'	61	\	'\u005C'	92
>	'\u003E'	62	]	'\u005D'	93
?	'\u003F'	63	^	'\u005E'	94
-	'\u005F'	95	o	'\u006F'	111
'	'\u0060'	96	p	'\u0070'	112
a	'\u0061'	97	q	'\u0071'	113
b	'\u0062'	98	r	'\u0072'	114
c	'\u0063'	99	s	'\u0073'	115
d	'\u0064'	100	t	'\u0074'	116
e	'\u0065'	101	u	'\u0075'	117
f	'\u0066'	102	v	'\u0076'	118
g	'\u0067'	103	w	'\u0077'	119
h	'\u0068'	104	x	'\u0078'	120
i	'\u0069'	105	y	'\u0079'	121
j	'\u006A'	106	z	'\u007A'	122
k	'\u006B'	107	{	'\u007B'	123
l	'\u006C'	108		'\u007C'	124
m	'\u006D'	109	}	'\u007D'	125
n	'\u006E'	110	~	'\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'

# Glossario

**Accesso casuale** La possibilità di accedere direttamente a qualsiasi valore senza dover leggere i valori che lo precedono.

**Accesso di pacchetto** Possibilità di accesso da parte dei metodi di classi contenute nel medesimo pacchetto.

**Accesso sequenziale** Accesso a valori in sequenza, senza saltarne alcuno.

**Accoppiamento** Il grado di mutua correlazione tra classi per effetto di dipendenze.

**Adattatore (stub)** Un metodo privo di funzionalità o con funzionalità minimali.

**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.

**Albero** Una struttura di dati costituita da nodi, ciascuno dei quali ha un elenco di nodi figli. Uno

dei nodi dell'albero ha la speciale funzione di nodo radice.

**Albero bilanciato** Un albero in cui ciascun sotto-albero ha un numero di discendenti sinistri circa uguale al numero di discendenti destri.

**Albero binario** Un albero in cui ogni nodo ha al massimo due nodi figli.

**Albero di ricerca binario** Un albero binario in cui ciascun nodo ha la proprietà di avere in tutti i discendenti sinistri un valore inferiore a quello memorizzato nel nodo stesso, e un valore superiore in tutti i discendenti destri.

**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 nell'invocazione di un metodo, oppure uno dei valori (*operandi*) su cui agisce un operatore.

**Array** Una raccolta di valori del medesimo tipo, memorizzati in posizioni di memoria contigue, a ciascuno dei quali si può accedere mediante un indice intero.

**Array (tipo di dato astratto)** Una sequenza ordinata di elementi che consente un efficiente accesso casuale mediante un indice intero.

**Array bidimensionale** Una disposizione tabulare di elementi, a ciascuno dei quali si accede 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.

**ArrayList** Una classe Java che realizza un array di oggetti che può ridimensionarsi dinamicamente (*vettore*).

**Assegnazione** Inserimento di un nuovo valore in una variabile.

**Asserzione** La dichiarazione 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)`.

**Astrazione** Il processo che identifica le caratteristiche essenziali di un blocco costitutivo di un programma, come una classe.

**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.

**Bit di segno** In un numero binario, il bit che indica se il numero è positivo o negativo.

**Blocco** Un gruppo di enunciati racchiusi tra parentesi graffe.

**Blocco annidato** Un blocco di enunciati contenuto all'interno di un altro blocco.

**Blocco try** Un blocco di enunciati che contiene clausole di elaborazione delle eccezioni. Un blocco `try` contiene almeno una clausola `catch` o una clausola `finally`.

**Breakpoint** Una posizione, all'interno di un programma, nella quale si desidera l'arresto del debugger, per ispezionare lo stato del programma in esecuzione controllata.

**Bucket** In una tabella hash, l'insieme di valori aventi il medesimo codice di hash.

**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.

**Campo di testo** Un componente dell'interfaccia grafica che consente all'utente di fornire testo in ingresso.

**Carattere** Una singola lettera, una cifra o un simbolo.

**Carattere di escape** In un testo, 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 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.

**Caratteristica privata** Una caratteristica che è accessibile solamente ai metodi della classe stessa o di una sua classe interna.

**Caratteristica protetta** Una caratteristica che è accessibile solamente ai metodi della classe stessa, di una sua classe interna, di una sua sottoclasse o di classi contenute nel medesimo pacchetto.

**Caratteristica pubblica** Una caratteristica che è accessibile da qualsiasi classe.

**Cartella (directory)** Una struttura di un *file system* che è in grado di contenere file o altre cartelle.

**Caso di prova “positivo”** Un caso di prova in cui ci si aspetta che il programma si comporti correttamente.

**Caso di prova “negativo”** Un caso di prova in cui ci si aspetta che il programma fallisca (ad esempio, collaudando con il valore  $-1$  un programma che calcola la radice quadrata nell’insieme dei numeri reali).

**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.

**Class path** L’insieme di cartelle e archivi in cui la macchina virtuale cerca i file di classi.

**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; 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.

**Clonazione** Esecuzione di una copia di un oggetto il cui stato può poi essere modificato in modo indipendente dall’oggetto originario.

**Coda** Un insieme di elementi che vengono estratti con una strategia “il primo entrato è il primo a uscire”.

**Coda prioritaria** Un tipo di dato astratto 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 della classe 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, per due diversi oggetti, una funzione di hash calcola il medesimo codice di hash.

**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 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.

**Contenitore** Un componente dell'interfaccia grafica che è in grado di contenere altri componenti e di presentarli insieme all'utente. Si parla di contenitore anche per una struttura di dati, come una lista, che può contenere una raccolta di oggetti.

**Contesto grafico** Una classe mediante la quale un programmatore può disegnare forme grafiche all'interno di una finestra o in un file.

**Copertura del collaudo** L'insieme delle istruzioni di un programma che vengono eseguite durante un collaudo.

**Costante** Un valore che non può essere modificato dal programma. In Java, le costanti vengono dichiarate mediante la parola riservata `final`.

**Costante numerica (o letterale)** Un valore costante che, all'interno di un programma, viene scritto esplicitamente sotto forma di numero, come `-2` oppure `6.02214115E23`.

**Costruttore** Un metodo che inizializza un oggetto appena creato.

**Costruttore predefinito** Un costruttore che viene invocato senza alcun parametro.

**Costruzione** Impostazione dello stato iniziale di un oggetto appena creato.

**CPU (Central Processing Unit)** Unità centrale di elaborazione, la parte di calcolatore che esegue le istruzioni in linguaggio macchina.

**Debugger** Un programma che consente all'utente l'esecuzione passo dopo passo di un altro programma, con la possibilità di interrompere l'esecuzione e di ispezionare le variabili, per agevolare l'identificazione degli errori.

**Diagramma sintattico** Una rappresentazione grafica di regole sintattiche.

**Directory** Vedi Cartella

**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 controllata (o a controllo obbligatorio)** Un'eccezione controllata dal compilatore. Tutte le eccezioni controllate devono essere dichiarate o gestite.

**Eccezione non controllata (o a controllo non obbligatorio)** Un'eccezione che non viene controllata dal compilatore.

**Editor** Un programma utilizzato per scrivere e modificare file di testo.

**Effetto collaterale** Un effetto visibile di un metodo, diverso dal valore restituito.

**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 composto** Un enunciato, come `if` o `while`, che è composto da più parti, che possono essere, ad esempio, una condizione e un corpo.

**Enunciato goto** Un enunciato che trasferisce il controllo d'esecuzione a un altro enunciato, identificato da un'etichetta. In Java non esiste l'enunciato `goto`.

**Enunciato semplice** Un enunciato costituito da un'unica espressione.

**Ereditarietà** La relazione "è un" 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 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 in compilazione).

**Errore in compilazione** Un errore che viene identificato durante la compilazione di un programma.

**Errore in esecuzione** Un errore che si verifica durante l'esecuzione di un programma sintatticamente corretto, portandolo ad agire in modo diverso da quanto previsto.

**Errore logico** Un errore presente in un programma sintatticamente corretto, che ne provoca un comportamento diverso dal previsto (un tipo di errore in esecuzione).

**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".

**Estensione** L'ultima parte del nome di un file, che ne specifica la tipologia. Ad esempio, l'estensione .java identifica i file di 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 binario** Un file i cui valori vengono memorizzati nella loro rappresentazione binaria e non possono essere letti come testo.

**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.

**Fine del file** Una condizione che risulta essere vera solamente quando tutti i caratteri di un file sono stati letti. Si noti che non esiste alcuno speciale "carattere di fine del file". Quando si digita sulla tastiera il contenuto di un file, può darsi che sia necessario inviare al sistema operativo un carattere speciale che indichi la fine del file, ma tale carattere non entra a far parte del file stesso.

**Finestra di shell** Una finestra che consente di interagire con il sistema operativo mediante comandi di tipo testuale.

**Firma di un metodo** Il nome di un metodo e i tipi dei suoi parametri.

**Flusso (stream)** Un'astrazione di una sequenza di byte da cui si possono leggere dati 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 di hash** Una funzione che calcola un valore di tipo 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 in cui l'utente fornisce dati in ingresso usando componenti grafici, come pulsanti, menu e campi di testo.

**Hashing** Applicazione di una funzione di hash a un insieme di oggetti.

**Heap** Un particolare albero binario bilanciato, usato per realizzare algoritmi di ordinamento e code prioritarie.

**Heapsort** Vedi Ordinamento con heap.

**HTML (Hypertext Markup Language)** Il linguaggio con cui vengono descritte le pagine Web.

**IDE (Integrated Development Environment)** Un ambiente di programmazione che contiene un editor, un compilatore e un debugger.

**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 "qualificato".

**Incapsulamento** L'occultamento dei dettagli realizzativi.

**Inizializzazione** Impostazione di una variabile a un valore ben preciso nel momento in cui viene creata.

**Insieme** Una raccolta non ordinata di elementi che consente di effettuare in modo efficiente l'aggiunta, l'eliminazione e la ricerca di elementi.

**Interfaccia** Un tipo di dato privo di variabili di esemplare, contenente soltanto costanti e metodi astratti.

**Interfaccia pubblica** Le caratteristiche di una classe (metodi, variabili e tipi interni) che sono accessibili agli utilizzatori.

**Internet** Un insieme di reti diffuse su tutta la Terra, costituite di apparati di commutazione e intradattamento e di calcolatori che usano un comune insieme di protocolli, che definiscono come gli utenti e i calcolatori della rete possono interagire reciprocamente.

**Interprete** Un programma che legge un insieme di codici ed esegue i comandi da essi specificati.

**Invariante di ciclo** Un'affermazione, relativa allo stato di un programma, che rimane vera ogni volta che gli enunciati del ciclo vengono eseguiti.

**Invocazione per riferimento** Un meccanismo di invocazione di metodo mediante il quale il metodo riceve l'indirizzo in memoria di una variabile fornita come parametro effettivo. L'invocazione per riferimento consente al metodo di modificare il contenuto della variabile originaria, in modo che le modifiche abbiano effetto anche dopo la terminazione dell'esecuzione del metodo.

**Invocazione per valore** Un meccanismo di invocazione di metodo mediante il quale il metodo riceve una copia del contenuto di una variabile fornita come parametro effettivo: Java usa solamente l'invocazione per valore. Se il tipo di una variabile parametro è una classe, il suo valore è un riferimento a un oggetto, per cui il metodo può manipolare tale oggetto, ma non può fare in modo che la variabile usata come parametro faccia riferimento a un diverso oggetto.

**Istanza** Vedi Esemplare.

**Iteratore** Un oggetto che è in grado di ispezionare tutti gli elementi presenti in un contenitore, quale 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.

**JDK** L'ambiente di sviluppo software per il linguaggio Java: contiene il compilatore Java e altri strumenti di sviluppo.

**JVM (Java Virtual Machine)** La macchina virtuale Java.

**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*.

**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 scripting** Un linguaggio di programmazione che privilegia la rapida progettazione di prototipi piuttosto della velocità di esecuzione o della facilità di manutenzione del codice.

**Lista** Una sequenza ordinata di elementi che possono essere scanditi in successione, con la possibilità di inserire e rimuovere elementi in ogni posizione in modo efficiente.

**Lista concatenata** Una struttura di dati che può contenere un numero arbitrario di oggetti, ciascuno dei quali viene memorizzato 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 dati che memorizza associazioni fra oggetti che fungono da chiave e oggetti che hanno il ruolo di valori.

**Mergesort** Vedi Ordinamento per fusione

**Metodo** Una sequenza di enunciati che ha un nome, può avere parametri formali 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 di esemplare** Un metodo avente un para-

metro implicito, cioè un metodo che viene invocato mediante un esemplare di una classe.

**Metodo generico** Un metodo con tipi parametrici.

**Metodo main** Il primo metodo che viene invocato quando viene eseguita un'applicazione Java.

**Metodo modificatore** Un metodo che modifica lo stato di un oggetto.

**Metodo predicativo** Un metodo che restituisce un valore booleano.

**Metodo ricorsivo** Un metodo che invoca se stesso con dati più semplici. Deve gestire i dati più semplici senza invocare se stesso.

**Metodo statico o di classe** Un metodo privo di parametro implicito.

**Mock** Vedi Oggetto semplificato.

**Nome qualificato** Un nome reso non ambiguo dal fatto che inizia con il nome di un pacchetto.

**Notazione O-grande** La notazione  $g(n) = O(f(n))$ , che indica che la funzione  $g$  ha un tasso di crescita rispetto a  $n$  limitato superiormente dal tasso di crescita della funzione  $f$ .

**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 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.

**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 postfisso** Un operatore unario che viene scritto dopo il suo argomento.

**Operatore prefisso** Un operatore unario che viene scritto prima del suo argomento.

**Operatore relazionale** Un operatore che confronta due valori, fornendo un risultato di tipo booleano.

**Operatore ternario** Un operatore dotato di tre argomenti. Java ha un solo operatore ternario, a ? b : c.

**Operatore unario** Un operatore che ha un solo argomento.

**Ordinamento con heap** Un algoritmo di ordinamento (*heapsort*) che inserisce all'interno di uno heap i valori da ordinare.

**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 differiscano 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 (*mergesort*) 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 sottosequenze.

**Ordinamento totale** Una relazione di ordinamen-

to mediante la quale ciascun elemento può essere confrontato con qualsiasi altro.

**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 utente privo di aspetti visibili; viene solitamente utilizzato per raggruppare altri componenti.

**Pannello dei contenuti** In Swing, quella porzione di frame che contiene i componenti dell'interfaccia utente.

**Parametro** Un elemento di informazione che viene comunicato a un metodo nel momento della sua invocazione. Ad esempio, nell'invocazione System.out.println("Hello, World!"), i parametri sono il parametro implicito System.out e il parametro esplicito "Hello, World!".

**Parametro effettivo** L'espressione fornita come valore per un parametro formale di un metodo da chi lo invoca.

**Parametro esplicito** In un metodo, un parametro diverso dall'oggetto con cui il metodo è stato invocato.

**Parametro formale** Nella definizione di un metodo, una variabile che, nel momento in cui il metodo viene invocato, viene inizializzata con il valore di un parametro effettivo.

**Parametro隐式的** 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** Definire espressioni che fungono da parametri effettivi di un metodo nel momento in cui questo viene invocato.

**Permutazione** Una disposizione degli elementi appartenenti a un insieme di valori.

**Pila** Una struttura di dati 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 di dati 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.

**Post-condizione** Una condizione che risulta essere vera al termine dell'invocazione di un metodo.

**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 interpretata come `a || (b && c)`.

**Pre-condizione** Una condizione che deve essere vera nel momento in cui viene invocato un metodo, per far sì che possa funzionare correttamente.

**Progetto** Un insieme di file contenenti codice sorgente e le loro dipendenze.

**Progettazione orientata agli oggetti** Progettazione di un programma definendo oggetti, le loro proprietà e le loro relazioni mutue.

**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 generica** Progettazione di componenti che possono essere riutilizzati in un'ampia gamma di situazioni.

**Programmazione visuale** Metodologia di programmazione che prevede la disposizione di elementi grafici all'interno di uno schema, impostandone il comportamento mediante la selezione di proprietà per ciascun elemento, scrivendo soltanto una modesta quantità di codice che funga da "collante" tra gli elementi grafici.

**Prompt** Una stringa 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.

**RAM (Random-Access Memory)** Circuiti elettronici interni a un calcolatore che sono in grado di memorizzare istruzioni e dati di programmi in esecuzione.

**Realizzazione di un'interfaccia** Realizzazione di una classe che definisce tutti i metodi specificati nell'interfaccia.

**Redirezione** Collegamento dell'ingresso o dell'uscita di un programma a un file, anziché alla tastiera o, rispettivamente, allo schermo.

**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 in base alla classe dell'oggetto che funge da parametro隐式.

**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 collocazione 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.

**Shadowing** Nascondere una variabile definendone un'altra con lo stesso nome.

**Shell** Un servizio del sistema operativo che consente all'utente di digitare righe di comandi per eseguire programmi e manipolare file.

**Sintassi** Regole che definiscono la composizione delle istruzioni in un particolare linguaggio di programmazione.

**Sistema embedded** Un sistema (processore, software e circuiteria di supporto) facente parte di un dispositivo diverso da un computer.

**Sistema operativo** Il software che esegue i programmi applicativi e fornisce loro servizi (come il *file system*).

**Sorgente di eventi** Un oggetto che è in grado di inviare ad altri oggetti notifiche relative a specifici eventi.

**Sottoclassa** Una classe che eredita variabili e metodi da una superclasse, con la possibilità di aggiungere variabili 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.

**Stub** Vedi Adattatore.

**Superclasse** Una classe da cui una classe più specifica (denominata sottoclasse) può ereditare.

**Swing** Un insieme di strumenti Java per la realizzazione di interfacce utente grafiche.

**Tabella hash** Una struttura di dati 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.

**throws** Indica il tipo di eccezioni a controllo obbligatorio che possono essere lanciate da un metodo.

**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 enumerativo** Un tipo di dato che può assumere un numero finito di valori, ciascuno dei quali è dotato di un proprio nome simbolico.

**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.

**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 codice 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 di cortocircuito** Valutazione di una parte 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 di tipo** Nella dichiarazione di un tipo generico, una variabile che viene sostituita da un tipo effettivo quando viene creato un 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 non inizializzata costituisce un errore di sintassi.

**Variabile parametro** In un metodo, una variabile che viene inizializzata con il valore di un parametro nel momento in cui il metodo viene invocato.

**Variabile statica** Una variabile, definita in una classe, di cui esiste una sola copia per l'intera classe e a cui possono accedere, eventualmente modificandone il valore, tutti i metodi della classe stessa.

**void** Una parola riservata che non indica alcun tipo o un tipo sconosciuto.

# Proprietà intellettuale delle immagini

## Capitolo 1

*Figura 1:* Intel Corporation

*Figura 2:* Intel Corporation

*Figura 3:* PhotoDisc, Inc./Getty Images

*Figura 4:* Intel Corporation

*Figura pag. 7:* Sperry Univac, Division of Sperry Corporation

## Capitolo 2

*Figura pag. 65:* Corbis Digital Stock

## Capitolo 3

*Figura pag. 118:* Punchstosk

## Capitolo 13

*Figura pag. 582:* Topham/The Image Works

# Indice analitico

I numeri di pagina che iniziano con W si riferiscono al Capitolo 16, disponibile sul sito web dedicato al libro.

## Simboli

`!=`, *Vedi* operatore relazionale  
`^`, 155  
`$`, 36  
`%`, *Vedi* operatore aritmetico  
`&`, *W9*, *Vedi anche* operatore booleano  
`&&`, *Vedi* operatore booleano  
`'`, 160-161  
`*`, *Vedi* operatore aritmetico  
`*/`, *Vedi* `/*`, `/**`  
`+`, 156, 463, *Vedi anche* operatore aritmetico  
`++`, 139  
`-`, *Vedi* operatore aritmetico  
`--`, 139  
`...`, 288  
`/`, *Vedi* operatore aritmetico  
`//`, 12  
`/*`, 14-15  
`/**`, 91-94  
`<, <=`, *Vedi* operatore relazionale

`<<`, 711-713  
`=`, *Vedi* assegnazione  
`==`, 179-183, *Vedi anche* operatore relazionale  
`>, >=`, *Vedi* operatore relazionale  
`>>, >>>`, 711-713  
`?:`, 179  
`@ensures`, 352  
`@param`, 92-93  
`@postcondition`, 352  
`@precondition`, 350  
`@requires`, 350  
`@return`, 92-94  
`@Test`, 374  
`[]`, 283-284  
`\, \\`, *Vedi* sequenza di escape  
`_`, 36  
`!, !!`, *Vedi* operatore booleano  
 $\Theta$  (Theta), 572-573  
 $\pi$  (pi), 261-263  
 $\Omega$  (Omega), 572-573

## A

`abstract`, 448-449  
`AbstractButton`, 431  
`AbstractSet`, 644  
accesso  
casuale, 4, 596, 614-617  
modalità di, *Vedi* modalità d'accesso sequenziale, 596, 616-617  
accoppiamento, 339-341, 386, 395-396  
`ActionEvent`, 405  
`ActionListener`, 405, 411-412, 414, 415  
Ada (linguaggio), 583  
ADT (abstract data type), *Vedi* tipo di dato astratto  
AI (Artificial Intelligence), 210-211, 422  
albero, 630, 639, 654  
altezza, 660-662, 673  
bilanciato, 660-662, 673  
binario, 654, 656, 669  
completo, 669  
di espressione, 667-668

- di ricerca binario, 654-665, 669, 673  
**heap**, *Vedi* heap  
 rosso-nero, 662  
 sbilanciato, 662  
 sintattico, 552-553  
 visita di, 665-668  
 alfabeto internazionale, 163  
 algoritmo, 21-29, 241-245, 302-309, 617, 619, 656  
 prestazione di un, 565-570, 576-579, 581, 586-587  
*Vedi anche* ordinamento, ricerca  
 Altair 8800, 372  
 altoparlante, 5-6  
 ambiente di programmazione, 15-17, 54, 106, 265-267, 373-375, 486  
 ambiguità, 24  
 ambito di visibilità, *Vedi* variabile, visibilità Analytical Engine, 582  
 and (**&&**), *Vedi* operatore booleano  
 Andersen, Marc, 78  
 annotazione, 374  
 ANSI (American National Standards Institute), 422, 618  
 API (Application Programming Interface), 49-53  
 Apple, 372-373  
 AppleScript, 474  
 applet, 8-9, 71-73, 365  
 applicazione, 49  
     grafica, 64, 410  
 architettura, 3-6, 8-9, 423  
 argomento, *Vedi* parametro  
     sulla riga dei comandi, 486-487  
**args**, 12, 486-487  
 Ariane, 518  
**ArithmetiException**, 503  
 ARPANET, 78  
 array, 282-287, 302-309, 312-319, 596, 617, 630, 642-644, 654, 659, 674, 682, W7-W8  
 algoritmi per, 302-309  
 bidimensionale, 260, 323-332  
 circolare, 620  
 di bucket, 643-644  
 di oggetti, 287  
 generico, W13  
 indice non valido, 284-285  
 indice valido, 283  
 inizializzazione, 282-283, 286  
**length**, 284, 295  
 list, *Vedi* vettore  
 lunghezza, 282  
 multidimensionale, 331  
 operatore [**1**], 283-284  
 paralleli, 287  
 riempito solo in parte, 300-301  
 variabile di tipo, 282  
**ArrayList**, 290-296, 588-589, 597, 601-602, 616-617, 634, 638-639, W2-W3, W10  
**add**, 290-292  
 classe generica, 290  
 costruttore, 290, 292  
**get**, 290-292, 501  
 in Java versione 7, 295-296  
 indice valido, 290-291  
**remove**, 292  
**set**, 291-292  
**size**, 291-292, 295  
**Arrays**, 307  
**binarySearch**, 587  
**copyOf**, 307-308  
**sort**, 581, 587-590  
**toString**, 312, 563  
 arrotondamento, 129, 132, 142  
     cast, 142  
     errore di, 129, 146  
 assegnazione, 38-41, 146  
**assert**, 350-351  
**AssertionError**, 350  
 asserzione, 350-351  
 AT&T, 255  
 auto-boxing, 296-297  
 auto-unboxing, 297  
 AWT (Abstract Windowing Toolkit), 52, 365, 404
- B**
- Babbage, Charles, 582-583  
 backslash, *Vedi* sequenza di escape  
 backup, 18-19  
 BASIC, 372  
 Basic Latin, 715-716  
 batch, file, 322  
 Bentley, Jon, 265  
 Berners-Lee, Tim, 78  
**BigDecimal**, 129-130  
**BigInteger**, 128, 130  
 binary search tree, *Vedi* albero  
     di ricerca binario  
 BIOS (Basic Input/Output System), 372  
 bit, 130-131  
 black box, 87, 209  
 blocco di enunciati, 175, 408  
 blocco di inizializzazione, 360-361  
 BlueJ, 54-55, 94, 106, 266, 373  
 Booch, Grady, 340  
 Boole, George, 203  
**Boolean**, 296  
**boolean**, 128, 203-206, 296  
**break**, 193-194, 254, 255-256  
 breakpoint, *Vedi* debugger, breakpoint  
 browser, 9, 71, 78, 93, 618  
 bucket, 643-644  
 buffer, 289  
 Buffon, Georges-Louis Leclerc de, 261  
 bug, 20, 276, 320  
 Burroughs, 65  
 bus, 5-6  
**Byte**, 296  
**byte**, 128, 142, 296  
**byte**, 128
- C**
- C (linguaggio), 194, 289, 302, 422  
 C++, 9-10, 147, 289, 348, 356, 359, 422, 618  
 calcolatrice, 621-624  
 calcolo automatico, 550-551  
 call stack, 528  
 callback, 393-398, 406  
 cancellazione dei tipi generici, W11-W13  
 canonica (espressione), 488, 492-493  
 carattere  
     backslash, *Vedi* sequenza di escape  
     di controllo, *Vedi* sequenza di escape  
     di escape, *Vedi* sequenza di escape  
     di sottolineatura, 36  
     di tabulazione, 177-178, 488  
     dollaro, 36  
     jolly, W10-W11  
     nuova riga (**\n**), 160, 483, 488-489  
     spazio, 36, 488  
     “spazio bianco”, 488, 490-491  
     supplementare, 161  
     virgolette, *Vedi* stringa  
     white space, 488, 490-491

- cartella (del file system), 17-18, 107, 533-537  
 di base, 368  
**case**, 193-194  
 caso di prova, 211-213  
 caso limite, 211  
 cast, 69, 142, 146, 389, 445, 465, 468, W13  
**catch**, 506-509, 511  
 CD (Compact Disc), 3  
 cerchio, 74-75  
 Cerf, Vinton, 78  
**char**, 128, 160-161, 296  
**Character**, 203, 296  
 isDigit, 203, 489  
 isLetter, 203  
 isLowerCase, 203  
 isUpperCase, 203-204  
 isWhiteSpace, 489  
**checked** (exception). *Vedi* eccezione  
 a controllo obbligatorio  
**chiave**, 634-640, 651  
 hardware, 685  
**chip**, 3, 372  
**CIA Fact Book**, 489  
**ciclicità**, 320  
**ciclo**, 247-253  
 annidato, 256-257  
 do, 230-231  
 e mezzo, 245, 253-254, 255  
 e ricorsione, 526  
 errore per scarto di uno, 228-229  
 esterno, 256-257  
**for**. *Vedi for*  
 infinito, 225, 228  
 interno, 256-257  
**while**, 222-225  
**cifra significativa**, 34  
**circuito integrato**, 3, 372  
**Class**, 467  
 getClass, 467-468  
**class**, 87  
**ClassCastException**, 503, W3  
**classe**, 11, 41-43, 338-339  
 astratta, 448-449  
 che realizza un'interfaccia, 384-385, 407  
 collaudo di unità, 106-108, 373-375  
 commento di documentazione, 92-93  
 concreta, 449  
 conversione a interfaccia, 388-389  
 di collaudo o di test, 107  
 di utilità, 338  
 dichiarazione, 90, 97  
 final, 449-450  
 generica, 290, W2-W11  
 gerarchia di ereditarietà, 430-432  
 "grezza", W11  
 immutable, 342  
 interna, 74, 398-401, 407-412, 602  
 interna anonima, 400-401  
 interna statica, 614  
 involucro, 296-297, W2  
 realizzazione di, 95-102  
**ClassNotFoundException**, 503  
**client/server**, 9  
**clipboard**, 403  
**clone**. *Vedi Object.clone*  
**Cloneable**, 471-472, W9  
**CloneNotSupportedException**, 471-472, 503  
**coda**, 619-620  
 prioritaria, 668-669  
**codice**  
 (di) hash, 138, 463, 642-644, 649-651  
 macchina, 6-8, 422  
 sorgente, 16-17, 482-483  
**coerenza**, 341-342  
**coesione**, 339-341  
**collaudo**, 20, 53-54, 102, 187, 201, 209, 211  
 a scatola bianca o aperta, 209  
 a scatola nera o chiusa, 209  
 caso di prova, 211, 212-213  
 caso limite, 211  
 copertura del, 209, 211, 212-213  
 di unità, 106-108  
 mock object, 401-404  
 pacchetto di prova, 211  
 regressivo, 320-322  
 trasparente, 209  
**collection**. *Vedi struttura dati*  
**Collections**, 588-589  
 binarySearch, 634  
 sort, 588-589  
**collisione**, 642  
**Color**, 76, 258-260, 630, 635-636  
 colori predefiniti, 76  
 getBlue, 259-260  
 getGreen, 259-260  
 getRed, 259-260  
**colore**, 76  
**commento**, 12, 14-15  
 di documentazione, 91-94  
**Comparable**, 587-590, 630, 639, 655, W8-W9, W11-W12  
**Comparator**, 590, 631, 639  
**compilatore**, 8, 15-17, 19-21  
**complemento a due**, 707-708  
**componente** (grafico), 67-71, 113-123, 411-414, 417, 431, 473  
 contenitore per, 411-414, 473  
**computer**, 2-3, 5-7  
**computer graphics**, 122  
**concatenazione** di stringhe, 156  
**condizione** (booleana), 203-209  
**condizione invariante**  
 di classe, 353-354  
 in un ciclo, 263-265, 353  
**conflitto** (di nomi di pacchetto), 367  
**confronto**  
 con null, 185  
 sequenze di, 191-194  
 tra numeri, 179-180  
 tra numeri in virgola mobile, 180-181  
 tra oggetti, 184  
 tra riferimenti, 184  
 tra stringhe, 181-183  
**connessione di rete**, 365, 403  
**contenitore**. *Vedi anche* struttura dati  
 di componenti grafici, 411-414, 473  
**contesto grafico**, 68-69, 76  
**continue**, 255-256  
**Control Data**, 65  
**conversione**, 142  
 tra sottoclasse e superclasse, 444-445  
**convenzione**  
 per identificatori, 37, 147  
 per spaziature, 144-145  
**copertura del collaudo**, 209, 211-213  
**copiatura**, 63-64  
 copia superficiale, 470  
**costante**, 132-138  
 dichiarazione, 134  
 in un'interfaccia, 387-388  
 nome, 134  
 statica, 359  
 universale, 138  
**costruttore**, 46-48, 89, 95-96, 101-102, 455-456, 460-461  
**corpo**, 89  
 di sottoclassi, 441-443  
 inizializzazione riferimenti in, 109

nome, 89  
**this()**, 112  
 costruzione, *Vedi* costruttore  
 CPU (Central Processing Unit), 3-8,  
     132  
 cursore, *Vedi* iteratore  
 CYC, 210

**D**

DARPA, 78, 211  
 database, 365, 422  
 debugger, 20, 265-277  
     breakpoint, 266-268, 273-275, 528  
     per un metodo ricorsivo, 528-529  
     single-step, 266-269, 275-276  
     step into/over, 268-269  
 debugging, *Vedi* debugger  
 decisione, *Vedi* if  
 De Morgan, Augustus, 208  
     legge di, 208-209  
 desktop, 402  
 Deutsch, L. Peter, 545  
 diagramma di flusso, 174, 175, 195, 205,  
     224, 230, 232  
 diagramma sintattico, 549, 552  
 diagramma UML, 340, 386, 396, 431-432,  
     452, 631, 635  
 Difference Engine, 582  
 dipendenza, 340-341, 386  
 diramazione, *Vedi* if  
 directory, *Vedi* cartella  
 disco ottico, 5-6  
 disco rigido (hard disk), 4, 6, 402, 482  
 disegnare, 67-71, 73-77  
 dispositivo periferico, 5-6  
 divisione intera, 139-140, 143-144  
 divisione per zero, 20  
**do**, 230-231  
 documentazione  
     API, 49-53, 55-57, 59  
     dell'interfaccia pubblica, 91-95  
 DOM (Document Object Model), 365  
 dongle, 685  
 DOS (Disk Operating System), 373  
**Double**, 296-297, 649  
     parseDouble, 157  
**double**, 34-35, 128-129, 142, 296-297  
 doubly linked list, 599, 617  
 Dr. Java, 54  
 duplicato, 630, 650

DVD, 5-6  
 dynamic method lookup, 391, 446

**E**

**ea (enableassertions)**, 351  
 eccezione, 20, 158-159, 350, 389, 501-519  
     a controllo obbligatorio, 504-506, 512  
     a controllo non obbligatorio, 504-506,  
         512  
     gerarchia di, 503  
     gestione di, 501, 503-504, 506-509  
     lancio di, 501-504, 508-509  
     progetto di, 512-513  
     squelching, 509  
 Eckert, J. Presper, 7  
 Eclipse, 266-267, 373  
 editor, 15-17, 178, 482  
 effetto collaterale, 185-186, 343-344,  
     346-347  
 efficienza, 617  
     della ricorsione, 539-545  
**Ellipse**, 74-75  
 ellisse, 73-75  
**else**, 175-176  
     sospeso, 197-198  
**enableassertions**, 351  
**endif**, 198  
 ENIAC, 7  
**Enum**, 472  
**enum**, 201-203, 472-473  
 enumerazione, 201-203, 472-473  
 enunciato, 8, 175  
     blocco di, 175, 408  
**EOFException**, 503  
**equals**, *Vedi* Object.equals  
 erasure, W11-W13  
 ereditarietà, 430-435  
     e genericità, W2, W10  
     e polimorfismo, 446-448  
     gerarchia di, 430-432, 452-462, 503  
     per personalizzare i frame, 473-475  
**Error**, 503-504  
 errore, 19-21  
     di arrotondamento,  
         *Vedi* arrotondamento  
     per scarto di uno, 228-229, 321  
 ESA (European Space Agency), 518  
 escape, *Vedi* sequenza di escape  
 esemplare, *Vedi* oggetto  
     variabile di, *Vedi* variabile di esemplare

espressione, 35, 549  
     algebrica, 621-624  
     booleana, 203-209  
     canonica, 488, 492-493  
     relazionale, 203  
     valutazione di, 549-558  
     valutazione "pigra" o di cortocircuito,  
         208  
 estensione (di Java), 67  
 etichetta, 255-256  
     grafica, 411  
 evento, 404-424, 619  
     adattatore per, 421-424  
     del mouse, 418-421  
     di temporizzazione, 415-417  
     notifica di, 405  
     ricevitore di, *Vedi* ricevitore di evento  
     sorgente di, 405, 414  
**Exception**, 503-504  
**extends**, 68-69, 423, 432-435, 450, 462,  
     512, W9-W11

**F**

**false**, 128, 203, 360  
 fattore, 549-558  
 Fericksen, Louis, 302  
**fi**, 198  
 Fibonacci, 539  
 FIFO (First In, First Out), 619, 668  
 figlio (nodo), 654  
**File**, 482, 485-486, 533-537  
     isDirectory, 534  
     listFiles, 534  
**file**, 17-18, 505, 533-537  
     batch, 322  
     chooser, 482, 485-486  
     di classe, 16-17  
     di testo, 482-484, 493-501  
     estensione del nome, 533  
     nome, 484  
     riga dei comandi, 486-487  
     script, 321  
     sorgente, 11, 15-17, 482  
     system, 17-18, 402, 482  
**FileNotFoundException**, 483, 503, 505,  
     507  
**final**, 134, 284, 365, 388, 409, 449-450  
**finally**, 509-511  
 finestra, 64, 402, 404  
     di dialogo, 168

- di console/shell/ terminale, 12, 15-16  
di tipo frame, 64-67, 70, 406, 411,  
473-476  
per la selezione di file, 482, 485-486
- finger**, 289
- Firefox**, 73
- firma (di un metodo), 88
- flag**, 205-206
- Float**, 296
- float**, 128-129, 142, 296
- floating point, *Vedi* numero  
in virgola mobile
- flusso**, 321
- foglia (nodo), 654
- folder**, *Vedi* cartella
- font**, 78, 402
- for**, 231-241  
**each**, 298  
esteso (o generalizzato), 298-300, 599,  
601-602, 632  
variabile definita in, 361-362
- foglio elettronico, 372
- formato libero, 15
- frame, *Vedi* finestra di tipo frame
- funzione, 140-142  
di hash, 642, 649-651
- G**
- garbage collector, 108, 603
- GE, 65
- genitore (nodo), 654
- gerarchia di ereditarietà, *Vedi* ereditarietà  
gerarchia di
- GNU, 78
- Gosling, James, 8-9, 422
- grafica al calcolatore, 122
- Graphics**, 68-69, 71-72
- Graphics2D**, 68-69, 71-72  
**draw**, 69, 74-77  
**drawString**, 69, 75-76  
**fill**, 76  
**setColor**, 76, 120
- grep**, 492-493
- Gutenberg, progetto, 78
- H**
- halt checker, 551
- halting problem, 550-551
- hard disk, *Vedi* disco rigido
- hash(ing), *Vedi*, codice hash, funzione  
di hash, tabella hash
- hashCode**, *Vedi* Object, hashCode
- HashMap**, 635-637, 639, 651
- HashSet**, 630-634, 637, 639, 653
- heap, 669-674, 680-682
- heapsort, 680-682
- Hewlett-Packard, 621
- Hoff, Marcian E. "Ted", 372
- Honeywell, 65
- Houston, Frank, 323
- HTML** (HyperText Markup Language),  
72-73, 91-95, 482, 618
- HTTP** (HyperText Transmission Protocol),  
618
- I**
- IBM, 65, 372
- icona, 431
- identificatore, 36-37
- IEEE, 133
- IETF (Internet Engineering Task Force),  
618
- if**, 174-176  
annidato, 194-197  
**else** sospeso, 197-198  
**if/else**, 175-176, 179, 211  
sequenza di confronti, 191-194
- IllegalArgumentException**, 502-504,  
512
- IllegalStateException**, 502-503, 606
- implementazione privata, 42
- implements**, 384-385
- import**, 52, 69, 74, 366-367
- importazione, 52, 366-367  
di un pacchetto, 367  
di una classe, 52, 366-367  
statica, 360
- incapsulamento, 86-87, 434, 469
- indentazione, 24
- indirizzo in memoria, 650-651
- IndexOutOfBoundsException**, 503
- informatica teorica, 551
- information hiding, *Vedi* incapsulamento
- ingresso standard, *Vedi* System.in
- inizializzazione (di variabile), 39-40  
blocco di, 359-361  
di esemplare, 109  
locale, 109  
parametro, 109
- statica, 359
- inner class, *Vedi* classe interna
- input standard, *Vedi* System.in
- InputMismatchException**, 491, 503-504
- insieme, 630-632, 637, 642, 651, 662
- instanceof**, 445-446, 468
- int**, 34-35, 128-129, 142, 296
- Integer**, 128, 296, 630-631  
**MAX\_VALUE**, 128  
**MIN\_VALUE**, 128  
**parseInt**, 132, 156, 490  
**toString**, 131
- Intel Corporation, 6, 133, 372, 568
- intelligenza artificiale, 210-211, 422
- interfaccia, 382-390, 403-404, 435  
conversione da classe a, 388-389  
costante in una, 387-388  
dichiarazione, 383  
di smistamento, *Vedi* callback  
**new**, 390  
parametrica, 590  
realizzazione di, 384-385, 407
- interfaccia pubblica, 42, 88-94, 95-97,  
99-100, 339, 438, 453-454, 459-460,  
616  
documentazione della, 91-95
- interfaccia utente grafica, 404, 487, 619
- interface**, 383
- interna, classe, *Vedi* classe interna
- Internet, 5, 9, 78, 289, 373
- Internet Explorer, 73, 78
- interprete, 474
- invariante, *Vedi* condizione invariante
- invocazione, *Vedi* metodo, invocazione di
- IOException**, 486, 503-505, 507
- ipertesto, 78
- ISO (International Organization  
for Standardization), 618
- istanza, *Vedi* esemplare
- istruzione, 2-3  
macchina, 6-8  
codifica di, 8
- Iterable**, 601-602
- Iterator**, 601-602, 631-632
- iteratore, 597-600, 605, 614-617, 631-632
- iterazione, *Vedi* ciclo
- J**
- Jacobson, Ivar, 340
- JApplet**, 71-73

Java, 8-11, 422-423  
 Community Process, 618  
 estensione, 67  
 versione 7, 295-296, 512, 637-638  
**java.applet**, 365  
**java.awt**, 52, 69, 365  
**java.awt.geom**, 74  
**java.io**, 365  
**java.lang**, 52, 365, 367  
**java.net**, 365  
**java.sql**, 365  
**java.util**, 52, 365, 597  
**javadoc**, 91-95, 350, 352  
**JavaScript**, 474  
**javax.swing**, 67, 365, 415  
**JButton**, 405-406, 410, 431  
**JCheckBox**, 431  
**JComponent**, 67-71, 115, 430-431  
 addMouseListener, 418-421  
 getHeight, getWidth, 115, 431  
 paintComponent, 68-69, 115-116,  
   416-417  
 repaint, 416-417, 419  
 setPreferredSize, 414  
**JDK** (Java Development Kit), 73, 474  
**JFileChooser**, 485-486  
**JFrame**, 64-67, 473-476  
 add, 70, 411  
 EXIT\_ON\_CLOSE, 66  
 setDefaultCloseOperation, 66  
 setSize, 66  
 setTitle, 66  
 setVisible, 66  
**JLabel**, 411, 417, 431  
**JOptionPane**, 168, 341  
**JPanel**, 411-414, 430-431  
**JRadioButton**, 431  
**JSwat**, 266  
**JTextArea**, 431  
**JTextComponent**, 431  
**JTextField**, 431  
**JToggleButton**, 431  
**JUnit**, 373-375  
**JVM** (Java Virtual Machine), 6-8, 17, 391,  
 446

**K**

Kahn, Bob, 78  
 key, *Vedi* chiave  
 key disk, 685

**L**

label, *Vedi* etichetta  
 labirinto, 619  
 Lenat, Douglas, 210  
 Lenovo, 372  
 letterale  
   carattere, 160-161  
   contenitore, 637  
   numerico, 34-35  
   stringa, 155  
 Leveson, 323  
 libreria (di Java), 9-10, 17, 422-423, 588,  
   597, 617-618, 620, 630, 637, 649, 662,  
   668, W2  
 documentazione API, 49-53  
 eccezioni, 502-503  
 enterprise/micro edition, 9  
 pacchetti, 365  
 Licklider, J. C. R., 78  
 LIFO (Last In, First Out), 619  
**Line2D**, 75  
 linea, 75  
 linea base, 75  
 linguaggio (di programmazione)  
   ad alto livello, 8, 422-423  
   di scripting, 474-475  
   funzionale, 423  
   macchina, 6-8, 422  
 linked list, *Vedi* lista concatenata  
**LinkedList**, 597-601, 616-617, 620, 634,  
   639, W2-W3, W10-W11, W14  
   addFirst/addLast 597  
   getFirst/getLast 597  
   listIterator, 597  
   removeFirst/removeLast 597  
 Linux, 322, 402, 568  
**List**, 634, 637-638, W14  
 lista, 617, 630  
   ad accesso casuale, *Vedi* vettore  
 lista concatenata, 596-617, 620, 630, 643,  
   654, 659, 669  
   come tipo di dato astratto, 614-617  
   doppiamente, 599, 617  
   realizzazione, 602-614  
   visita, 597  
 listener, *Vedi* ricevitore  
**ListIterator**, 597-602, 632, W10-W11  
 add, 599-600, 632  
 hasNext/hasPrevious, 598-600  
 next, 598-600

previous, 598-600, 632  
 remove, 599-600

locale, *Vedi* variabile locale

**Logger**, 214

logging, *Vedi* tracciamento

**Long**, 296

**long**, 128, 142, 296

Lovelace, Ada Augusta, 582-583

Lukasiewicz, Jan, 621

**M**

macchina di Turing, 550-551

macchina virtuale Java, *Vedi* JVM

Macintosh / Mac OS, 10, 322, 373, 403,  
 474

**main**, metodo, 11-14, 21, 106-107, 356,  
 475-476, 483, 486-487

mainframe, 65

maiuscolo/minuscolo, 15, 37

**MalformedURLException**, 503

**Map**, 634-638, 640

mappa, 634-636, 638-640, 642, 651, 662

Mark II, 276

**Math**, 134, 141, 147

E, 134

PI, 134

pow, 140-143

round, 142, 146

sqrt, 140-143, 147, 355

matrice, *Vedi* array bidimensionale

Mauchly, John, 7

max-heap, *Vedi* heap

memoria, 4-6

ad accesso casuale, 4

disponibile, 504

locazione di, *Vedi* variabile

principale, 4

RAM, 4, 402

rimovibile, 5

ROM, 372

secondaria, 4

virtuale, 402

menu, 102-106, 404

mergesort, 573-579, 588

metodo, 41-43

astratto, 383, 449

ausiliario, 538-539

commento di documentazione, 92-93

con numero di parametri variabile, 288

corpo, 88, 108

- d'accesso, 48-49, 96-97, 342, 469  
 della superclasse, 439  
 di classe, *Vedi* metodo statico  
 di esemplare, 354, 432  
 dichiarazione, 97  
 effetto collaterale, 343-344, 346-347  
 ereditato, 432-434  
**final**, 450  
 firma, 88  
 generico, W7-W12  
 invocazione di, 12-13, 43-45, 108-110, 348-349  
 modificatore, 48-49, 96-97, 342, 353  
 nome, 89  
 parametro di, 43-45, 89  
 predicativo, 203-204  
 ricerca dinamica del, 391, 446  
 ricorsivo, 525-528, 538-539  
 sovraccarico, 43, 440  
 sovrascritto, 437-441, 450-451, 453, 458-459, 462-467  
 statico, 147-148, 354-357  
 valore restituito da, 44-45, 89, 92-93  
 variabile locale, 108-109
- microfono, 5-6  
 microprocessore, 372  
 Microsoft, 73, 78, 200-201, 373, 618  
     Office, 474  
     Outlook, 475  
 Miller, Barton P., 302  
 min-heap, *Vedi* heap  
 MITS Electronics, 372  
 mock object, 401-404  
 modalità d'accesso, 84-85, 89  
     di pacchetto, 369-370, 387, 451  
     più ristretta in sottoclasse, 450-451  
     protected, 451, 471  
 monitor, 5-6  
 Mosaic, 78  
 motherboard, 5-6  
 mouse, 5-6, 404, 418-421  
     listener, *Vedi* ricevitore di eventi del mouse  
**MouseAdapter**, 423-424  
**MouseListener**, 418-421, 423-424  
 multitasking, 323, 402
- N**
- NASA, 518  
 Naughton, Patrick, 8
- NCR, 65  
 NCSA, 78  
 Netscape, 78  
**new**, 46-48, 95-96, W12-W13  
     costruzione di array, 282  
     costruzione di eccezione, 502  
 newline, *Vedi* carattere nuova riga  
 Nicely, Thomas, 133  
 nome di dominio, 367-368  
**NoSuchElementException**, 490, 503, 505, 507, 598  
**not (!)**, *Vedi* operatore booleano  
 notazione  
     algebrica, 621-624  
     polacca inversa (RPN), 621-624  
     scientifica, 34  
**Notepad**, 482  
**null**, 109, 185, 284, 286, 360, 504, 506, 606, 636, 655-656  
**NullPointerException**, 503-504  
**NumberFormatException**, 490, 503-504, 507  
 numero, 34, 128-129, 382  
     binario, 129, 130-132, 705-707  
     casuale, 260-263  
     conversione, 130-132, 705-707  
     decimale, 129  
     esadecimale, 709  
     frazionario, 131  
     grande, 130  
     in complemento a due, 707-708  
     in notazione scientifica, 490  
     in virgola mobile, 34, 128-129, 131-133, 490, 708-709  
     intero, 34, 128-129  
     magico, 138, 493  
     pseudo casuale, 261-262  
     sistema di numerazione, 130-132, 705-709  
     triangolare, 524-527
- O**
- O-grande, 570, 572-573, 578, 581, 586-587, 617, 630, 644, 654, 673, 680-682  
**Object**, 367, 394-395, 444, 462-472, 590, 602, 650-651, 662, W2, W11-W13  
     clone, 462, 465-466, 469-472  
     equals, 462, 464-465, 467-469, 639-640, 650-651  
**hashCode**, 630, 639-640, 642, 649-651, 653-654  
**toString**, 462-464, 466-467  
 oggetto, 12, 35, 41-43, 46-47, 84-86, 147, 449, 463-464, 649  
 anonimo, 400  
 confronto con null, 185  
 confronto mediante equals, 184, 464-465  
 di tipo interfaccia, 390  
 immutabile, 342  
 riferimento a, *Vedi* riferimento semplificato (mock), 401-404  
 stato di un, 84-86  
 variabile di esemplare, 84-86
- Omega ( $\Omega$ ), 572-573  
**omg.w3c.dom**, 365  
 open source, 474  
 operatore, 35, 621-624, 701-702  
     aritmetico, 35, 138-144, 146  
     bit a bit, 711-713  
     booleano, 204-205, 207-208  
     condizionale, 179  
     decremento, 139  
     di assegnazione, 38-41, 146  
     di concatenazione, 156, 463  
     di selezione, 179  
     di uguaglianza, 179-183  
     divisione intera, 139-140, 143-144  
     incremento, 139  
     precedenza o priorità, 180, 204  
     relazionale, 176, 179-181, 206-207  
 operazione, *Vedi* operatore  
 or(||), *Vedi* operatore booleano  
 ordinamento, 562-581, 654  
     alfabetico, 182  
     heapsort, 680-682  
     lessicografico, 182  
     per fusione, 573-579, 588  
     per inserimento, 563, 571-572  
     per selezione, 562-570  
     quicksort, 579-581  
**OutOfMemoryError**, 504  
 output standard, *Vedi* System.out  
 overflow, 128, *Vedi anche* stack  
     overflow  
 overload(ed), *Vedi* metodo  
     sovraccarico  
 override, overridden, *Vedi* metodo  
     sovrascritto

**P**

pacchetto (di libreria), 10, 365-371  
 di default, 366  
 importazione di, 52, 366-367  
 localizzazione nel file system, 368  
 nome, 367-368  
 predefinito, 366  
 pacchetto di prove, 211, 320  
**package**, 366  
 package, *Vedi* pacchetto  
 palindrome, 528-533  
 pannello (grafico), 411-414  
 parametro, 12, 43-45, *Vedi anche* variabile  
 parametro  
 commento di documentazione, 92-93  
 del metodo **main**, 486-487  
 esplicito, 43, 108  
 di costruzione, 46-48  
 implicito, 43, 57, 110-112, 447  
 in numero variabile, 288  
 modifica di, 345-346, 347  
 non valido, 502  
 parola riservata, 36, 703-704  
 parte frazionaria, 34  
 Pascal, 422  
 password, 402  
 pattern, 488, 492  
 PC, *Vedi* Personal Computer  
 Pentium, 6, 133  
 permutazione, 545-549  
 Personal Computer, 372-373  
 pi (greco,  $\pi$ ), 261-263  
 pila, 619-624  
   di esecuzione, 619  
 pirateria del software, 685  
 pivot, 580-581  
 pixel, 66, 258-260  
**Point2D**, 75  
 polimorfismo, 390-391, 446-448  
 portatile, 9-10  
 portabilità, 9-10  
 post-condizione, 349-353  
 posta elettronica, 78, 368  
 PostScript, 545  
 pre-condizione, 349-354  
 precisione  
   doppia, 34, 74, 128  
   singola, 74, 128  
 prestazione, *Vedi* O-grande, algoritmo  
 (prestazione di un)

**Q**

**Queue**, 620  
 queue, *Vedi* coda  
 quicksort, 579-581, 587

**R**

radice (nodo), 654  
 RAM (Random Access Memory), 4, 402  
**Random**, 52, 261-262  
   **nextDouble**, 261  
   **nextInt**, 261-262  
 random access, *Vedi* accesso casuale  
 raw type, W11  
 RCA, 65  
 Reagan, 65  
 record, 489  
**Rectangle**, 46-52, 62-64, 69-70, 184, 355, 630  
   **getHeight/getWidth**, 48  
   **getX/getY**, 48  
   **setSize**, 50-51  
   **translate**, 48, 53  
 redirezione, 321  
 regressivo, collaudo, 320-322  
 regular expression, 488, 492-493  
 relazione d'ordine totale, 588  
 rete di calcolatori, 5, 78  
 rettangolo di delimitazione, 74-75  
**return**, 86, 254, 351  
 RFC (Request For Comment), 618  
 RGB, 76  
 ricerca, 581-587  
   binaria o per bisezione, 584-587, 596, 617, 654  
   lineare o sequenziale, 581-584, 596, 630  
   dinamica del metodo, 391, 446  
 ricevitore di evento, 404-424  
   adattatore per, 421-424  
   del mouse, 418-421  
   associazione di, 413  
 ricorsione, 524-526, 619  
   efficienza della, 539-545  
   infinita, 438, 527-528  
   mutua, 549-558  
 riferimento, 62-64, 348-349, 449, 464-465, *Vedi anche* oggetto  
   a interfaccia, 634  
   ad array, 282  
   condivisione di, 342  
   confronto con **null**, 185  
   conversione da classe a interfaccia, 388-389  
   conversione da sottoclassse a superclasse, 444-445

- conversione da superclasse  
a sottoclasse, 445  
inizializzazione in costruttore, 109  
**null**, *Vedi null*  
ripristino (dal backup), 19  
riutilizzo del codice, 382-385, 444  
ROM (Read-Only Memory), 372  
RPN (Reverse Polish Notation), 621-624, 667-668  
Rumbaugh, James, 340  
runtime stack, *Vedi* pila di esecuzione  
**RuntimeException**, 503-504, 512
- S**
- Safari, 73  
Scala, 423  
Scanner, 161-162, 482, 484-486, 489-492  
  **hasNext**, 204, 350, 488, 492  
  **hasNextDouble**, 204, 491  
  **hasNextInt**, 204, 490-491  
  **next**, 162, 350, 482, 488-489, 492, 507  
  **nextDouble**, 161, 482, 490-492  
  **nextInt**, 161, 482, 490-492, 504  
  **nextLine**, 162, 482, 485, 489, 491-492  
  **useDelimiter**, 488, 492-493  
scatola nera, 87, 209  
scheda di memoria, 5  
scheda madre o principale, 5-6  
schema UML, *Vedi* diagramma UML  
Scheme, 618  
schermo, 5-6  
scope, *Vedi* variabile, visibilità  
scorrimento (bit a bit), 711-713  
script, 322, 474-475  
SDK (Software Development Kit), Java, 73, 474  
sentinella, 244-245  
separatore  
  decimale, 34  
  delle migliaia, 34  
sequenza, *Vedi* array  
  di caratteri, *Vedi* stringa  
  di escape, 159-160, 484, 715-716  
**Set**, 630-634, 636-637, 644, 653  
  **add**, 631  
  **addAll**, 637  
  **contains**, 631, 654  
  **remove**, 631  
shadowing, 364  
shell, 322, 474, *Vedi* finestra di terminale
- Short, 296  
short, 128, 296  
sicurezza, 9  
simulazione, 260-263  
single-step, *Vedi* debugger, single-step  
sistema di numerazione, *Vedi* numero  
sistema operativo, 372-373, 402-403, 486  
smart card, 9  
So, Bryan, 302  
sorgente  
  codice, 16-17, 482-483  
  di eventi, 405, 414  
  file, 11, 15-17, 482  
sort(ing), *Vedi* ordinamento  
sottoclasse, 430-451  
  costruttore di, 441-443  
  dichiarazione di, 432-435  
sottoinsieme, 436  
sottostringa, 157-158  
source, *Vedi* sorgente  
sovraffaccarico, *Vedi* metodo sovraccarico  
sovrascritto, *Vedi* metodo sovrascritto  
SPARC, 6  
spreadsheet, 372  
SQL (Structured Query Language), 365  
Stack, 620, 622-624  
stack, *Vedi* pila  
  overflow, 528  
stampante, 5-6, 402, 619  
standardizzazione, 618  
Stanley, 211  
static, 12, 134, 147, 354-361, 388, 614, W14  
statico, *Vedi* metodo statico, variabile  
statica  
step (into o over), *Vedi* debugger, step  
**String**, 13, 34, 36, 155-156, 630-631  
  **charAt**, 161, 489, 492  
  classe final, 450  
  classe immutabile, 342  
  **compareTo**, 182, 589  
  **equals**, 181-182  
  **equalsIgnoreCase**, 181-182  
  **format**, 168  
  **hashCode**, 642  
  **length**, 42, 48, 155-156, 295  
  **replace**, 44-45  
  **substring**, 157-158, 165, 489  
  **toUpperCase**, 42, 49, 342  
  **trim**, 52, 489-490  
stringa, 13, 155-158, 160-161
- concatenazione, 156  
confronto, 181-183  
conversione in numero, 156-157  
ordinamento, 182  
posizioni in una, 157  
sottostringa, 157-158  
vuota, 156  
**StringIndexOutOfBoundsException**, 158  
struttura (di) dati, 596-624, 630-680  
Sun Microsystems, 6, 8, 65, 73, 618  
**super**, 439-443, 467, 513, W10-W11  
superclasse, 430-449  
  universale, *Vedi* Object  
superinsieme, 436  
Swing, 67, 365, 417, 430-431, 485  
**SwingConstants**, 387-388  
switch, 193-194  
**System**, 52  
  **currentTimeMillis**, 565-567  
  **exit**, 168  
  **in**, 161  
  **out**, 12, 34, 41-42, 213-214, 505-506
- T**
- tabella, *Vedi* array bidimensionale  
tabella hash, 630, 642-650, 659  
  con bucket, 643-644  
tabulazione, *Vedi* carattere di tabulazione  
tar, 302  
tastiera, 5-6  
Tcl (tool control language), 474  
TCP/IP, 78  
telefono cellulare, 9  
temporizzazione, evento di, 415-417  
termine, 549-558  
test suite, *Vedi* pacchetto di prove  
test(ing), *Vedi* collaudo  
Therac-25, 323  
Theta ( $\Theta$ ), 572-573  
**this**, 57, 110-112, 363-364, 414, 438, 447  
thread, 168, 620  
throw, 501-502  
**Throwable**, 503, 513  
  **printStackTrace**, 507  
throws, 483, 486, 505-506, 515  
Thrum, Sebastian, 211  
Timer, 415-417  
tipo (di dato), 34-35  
astratto, 614-617, 619-620, 668-669  
booleano, 203

enumerativo, 201-203, 472-473  
 interfaccia, *Vedi* interfaccia  
 numerico, 128-129  
 parametrico, W2-W3, W7-W8,  
     W11-W14  
 primitivo, *Vedi* primitivo, tipo  
     di dato  
 vincolo per, W8-W11  
 token, 552-557  
**toString**, *Vedi* Object, **toString**  
 trabocco, *Vedi* overflow  
 tracciamento (dell'esecuzione), 213-214  
     di metodo ricorsivo, 528  
 transistor, 3, 7  
 trasferibile, *Vedi* portabile  
 trasferibilità, *Vedi* portabilità  
**TreeMap**, 635-637, 639-640  
**TreeSet**, 630-632, 639  
**true**, 128, 203  
**try**, 506-512  
 tubo a vuoto, 7  
 Turing, Alan, 550-551  
 Turner, 323  
 type erasure, W11-W13

**U**

UML (Unified Modeling Language),  
     340  
 unchecked (exception), *Vedi* eccezione  
     a controllo non obbligatorio  
 Unicode, 128, 160-161, 163, 715-716  
 unit testing, *Vedi* collaudo di unità  
 unità centrale (di elaborazione),  
     *Vedi* CPU  
 Univac, 65  
 UNIX, 10, 289, 302, 322, 368, 492  
**UnknownHostException**, 503  
**URL**, 486  
 uscita standard, *Vedi* System.out  
 utente, 402, 404  
**UTF-16**, 161

**V**

valore, 34-35  
     in una mappa, 634-636  
     iniziale, 36-38  
     oggetto, 47, 62-64  
     restituito, *Vedi* metodo, valore  
         restituito da  
         riferimento, 47  
         sentinella, 244-245  
 valutazione, *Vedi* espressione, valutazione  
 valvola termoionica, 7  
 van der Linden, Peter, 194, 255  
 variabile, 8, 35-38  
     array, 282  
     booleana, 205-206  
     copia di, 63-64  
     costante, *Vedi* costante  
     definita in un ciclo **for**, 361-362  
     dichiarazione, 36-38  
     di esemplare, *Vedi* variabile  
         di esemplare  
     di tipo, W2, W4-W5, W10  
     locale, *Vedi* variabile locale  
     mettere in ombra una, 364  
     nome, 36-38  
     numerica, 63  
     non inizializzata, 39  
     parametro, 89, 108-109  
     statica, 357-361, 409  
     visibilità, 239, 361-365  
 variabile di esemplare, 84-86, 111, 358,  
     384, 432, 454-455, 460, 602-603,  
     650  
     della classe esterna, 614  
     della superclasse, 433-434, 436-438  
     di tipo generico, W4  
     dichiarazione, 84-85  
     final, 409, 411  
     individuazione, 95, 101  
     inizializzazione, 109  
     mettere in ombra, 436-437  
     nome, 85

tipo, 85  
 visibilità, 363, 408  
*Vedi anche* encapsulamento

variabile locale, 108-109  
     final, 409  
     inizializzazione, 109  
     visibilità, 362, 408  
 vettore, 290-296, 302-309, 312-319, 620,  
     674  
     algoritmi per, 302-309  
     come tipo di dato astratto, 614-617  
 vincolo per tipo parametrico, W8-W11  
 virgola mobile, *Vedi* numero  
     in virgola mobile  
 virus, 289, 475  
 visibilità, *Vedi* variabile, visibilità  
**VisiCalc**, 372  
 visita (di albero), 665-668  
 Visual Basic Script, 474-475  
**void**, 89, 91  
 voto elettronico, 114-115

**W**

W3C (World Wide Web Consortium),  
     618  
 web, *Vedi* WWW  
**while**, 222-225  
 white-box testing, 209  
 white space, 488, 490-491  
 wildcard, W10-W11  
 Wilkes, Maurice, 276  
**Window**, 370  
 Windows, 10, 167, 322, 368, 403, 482,  
     492, 618  
 worm, 289  
 wrapper (classe involucro), 296-297, W2  
 WWW, World-Wide Web, 9-10, 78,  
     486, 618

**X**

XML, 365