Bruce Eckel

# Thinking in Java

## Tecniche avanzate

Quarta edizione

Copyright © 2006 Pearson Education Italia S.r.l.
Via Fara, 28 – 20124 Milano
Tel. 02/6739761 Fax 02/673976503
E-mail: hpeitalia@pearson.com
Web: http://hpe.pearsoned.it

*Authorized translation from the English language edition, entitled:* ***THINKING IN JAVA, 4th Edition, 0131872486*** *by Eckel, Bruce, published by Pearson Education, Inc, publishing as Prentice Hall PTR Copyright © 2006.*

*All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc*
*Italian language edition published by Pearson Education Italia Srl, Copyright © 2005*

Le informazioni contenute in questo libro sono state verificate e documentate con la massima cura possibile. Nessuna responsabilità derivante dal loro utilizzo potrà venire imputata agli Autori, a Pearson Education Italia o a ogni persona e società coinvolta nella creazione, produzione e distribuzione di questo libro.

I diritti di riproduzione e di memorizzazione elettronica totale e parziale con qualsiasi mezzo, compresi i microfilm e le copie fotostatiche, sono riservati per tutti i paesi.
LA FOTOCOPIATURA DEI LIBRI È UN REATO.
L'editore potrà concedere a pagamento l'autorizzazione a riprodurre una porzione non superiore a un decimo del presente volume. Le richieste di riproduzione vanno inoltrate ad AIDRO (Associazione Italiana per i Diritti di Riproduzione delle Opere dell'Ingegno), Via delle Erbe, 2 - 20121 Milano - Tel. e Fax 02/80.95.06.

Traduzione: Georges Piriou, Marco Tripolini
Revisione tecnica: Georges Piriou, Marco Tripolini
Realizzazione editoriale: Art Servizi Editoriali - Bologna
Grafica di copertina: Sabrina Miraglia

Stampa: Presscolor - Milano

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

ISBN 13: 978-8-8719-2304-8
ISBN 10: 88-7192-304-9

Printed in Italy

1a edizione: settembre 2006

## Struttura dell'opera

| | *Vol. 1* | *Vol. 2* | *Vol. 3* |
|---|---|---|---|
| **Prefazione** | • | • | • |
| **Introduzione** | • | • | • |
| **Introduzione agli oggetti** | • | | |
| **Tutto è un oggetto** | • | | |
| **Gli operatori** | • | | |
| **Il controllo dell'esecuzione** | • | | |
| **Inizializzazione e cleanup** | • | | |
| **Controllo di accesso** | • | | |
| **Riutilizzo delle classi** | • | | |
| **Polimorfismo** | • | | |
| **Le interfacce** | • | | |
| **Le classi interne** | • | | |
| **Come contenere gli oggetti** | • | | |
| **Gestione degli errori con le eccezioni** | | • | |
| **Le stringhe** | • | | |
| **Informazioni sui tipi** | | • | |
| **Generici** | | • | |
| **Array** | | • | |
| **Ancora sui contenitori** | | • | |
| **Input/Output** | | • | |
| **Tipi enumerativi** | | • | |
| **Annotazioni** | | • | |
| **La concorrenza** | | | • |
| **Interfacce grafiche (GUI)** | | | • |
| **Supplementi** | • | • | • |
| **Risorse** | • | • | • |

*L'edizione italiana presenta, rispetto a quella anglosassone, alcune importanti modifiche, che hanno portato alla suddivisione dell'opera originale in tre volumi e ad una parziale riorganizzazione dei contenuti. Lo schema qui sopra riportato illustra sinteticamente la struttura di questa edizione. Sono nati così tre testi autonomi, che speriamo rendano più agevole la consultazione e consentano anche una migliore fruibilità dei contenuti.*

*L'Editore*

# Indice

| | |
|---|---|
| **Prefazione** | **XV** |
| Java SE5 e SE6 | XVII |
| Java SE6 | XVII |
| La quarta edizione | XVIII |
| Modifiche | XVIII |
| Note sulla grafica di copertina | XX |
| Ringraziamenti dell'autore | XXI |
| **Introduzione** | **XXV** |
| Prerequisiti | XXVI |
| Imparare Java | XXVI |
| Obiettivi | XXVII |
| Imparare da questo libro | XXVIII |
| Documentazione JDK in HTML | XXIX |
| Esercizi | XXIX |
| Fondamenti di Java | XXX |
| Codice sorgente | XXX |
| Standard di codifica | XXXIII |
| Errori | XXXIII |
| **Gestione degli errori con le eccezioni** | **1** |
| Concetti fondamentali | 2 |
| Eccezioni di base | 3 |
| Argomenti delle eccezioni | 5 |
| Come intercettare un'eccezione | 5 |
| Blocco try | 6 |
| Gestori di eccezioni | 6 |
| Confronto tra interruzione e ripresa | 7 |
| Generazione di eccezioni personalizzate | 8 |
| Eccezioni e log | 11 |
| Specifiche di eccezione | 16 |
| Intercettare qualsiasi eccezione | 17 |
| Tracciamento dello stack | 20 |
| Eccezioni sollevate ripetutamente | 21 |

Concatenamento di eccezioni 25
Eccezioni Java standard 29
Caso speciale: RuntimeException 30
Esecuzione delle operazioni di cleanup con finally 32
A che cosa serve finally? 34
Utilizzo di finally durante un return 38
Attenzione all'eccezione perduta 39
Limiti delle eccezioni 42
Costruttori 46
Corrispondenza delle eccezioni 53
Tecniche alternative 54
Storia 56
Prospettive 58
Passaggio delle eccezioni alla console 61
Conversione delle eccezioni da controllate a non controllate 62
Guida di riferimento alle eccezioni 65
Riepilogo 66

**Informazioni sui tipi** **67**

L'esigenza di RTTI 67
L'oggetto Class 70
Letterali di classe 77
Riferimenti generici di classi 80
Nuova sintassi di cast 84
Controlli pre-casting 85
Utilizzo dei letterali di classe 94
Operatore instanceof dinamico 96
Conteggio ricorsivo 98
Factory registrate 100
Equivalenza di instanceof e Class 105
Riflessione: informazioni di classe a runtime 107
Un estrattore dei metodi di classe 109
Proxy dinamici 113
Oggetti null 119
Oggetti mock e stub 128
Interfacce e informazioni di tipo 128
Riepilogo 136

**Generici** **139**

Confronto con C++ 141
Generici semplici 141
Una libreria di tuple 144
Una classe di stack 148
RandomList 149
Interfacce generiche 150

Metodi generici 155
Sfruttare la deduzione del tipo 157
Specifica di tipo esplicita 159
Elenchi di argomenti variabili e metodi generici 160
Un metodo generico da utilizzare con i Generator 161
Un Generator multiutilizzo 162
Semplificare l'utilizzo delle tuple 164
Un programma di utilità per Set 166
Classi interne anonime 171
Costruzione di modelli complessi 174
Mistero della cancellazione 177
Approccio di C++ 179
Compatibilità in vista della migrazione 182
Problema della cancellazione 184
Azione sui limiti 186
Compensazione della cancellazione 191
Creazione di istanze di tipi 193
Array di generici 197
Limiti 203
Metacaratteri 208
Quanto è "intelligente" il compilatore? 212
Controvarianza 215
Metacaratteri senza limiti 218
Conversione dell'intercettazione 226
Problemi 227
Nessun primitivo come parametro di tipo 227
Implementare interfacce parametrizzate 230
Casting e avvertimenti del compilatore 231
Sovraccarico 234
Classe di base che dirotta un'interfaccia 235
Tipi autolimitati 236
Generici stranamente ricorsivi 236
Autolimitazione 238
Covarianza degli argomenti 241
Sicurezza di tipo dinamica 246
Eccezioni 247
Mixin 250
Mixin in C++ 250
Mixin con le interfacce 252
Utilizzo del modello Decorator 254
Mixin con proxy dinamici 257
Latent typing 259
Il latent typing non compromette lo strong typing 263
Compensazione alla mancanza di latent typing 265
Riflessione 265
Applicare un metodo a una sequenza 267
Come fare quando non si dispone dell'interfaccia corretta 271

Simulare il latent typing mediante gli adattatori 273
Strategia degli oggetti funzione 277
Riepilogo: il casting è davvero così sconveniente? 285
Ulteriori letture 288

## Array 289

Perché gli array sono speciali 289
Gli array sono oggetti di prima classe 292
Restituzione dell'array 296
Array multidimensionali 298
Array e generici 303
Creazione dei dati di prova 306
Arrays.fill() 307
Generatori di dati 308
Creazione di array con i generatori 315
Utility per Arrays 321
Copia di array 322
Confronto di array 324
Confronto tra elementi di array 325
Ordinamento di array 330
Ricerca in array ordinati 331
Riepilogo 334

## Ancora sui contenitori 339

Tassonomia completa dei contenitori 339
Popolare i contenitori 341
Una soluzione con Generator 342
Generatori di Map 345
Utilizzo delle classi astratte 349
Funzionalità di Collection 360
Operazioni facoltative 364
Operazioni non supportate 365
Funzionalità di List 368
Set e ordine di archiviazione 373
SortedSet 378
Code 379
Code con priorità 381
Deque 383
Approfondimenti su Map 384
Prestazioni 387
SortedMap 391
LinkedHashMap 393
Hashing e codici hash 394
Approfondimenti su hashCode() 399
Ottimizzazione dell'hashing 403

Sovrascrittura del metodo hashCode() 408
Scegliere un'implementazione 415
Struttura per test prestazionali 416
Valutazione delle List 421
Rischi impliciti nei microbenchmark 430
Valutazione dei Set 432
Valutazione delle Map 434
Fattori che influenzano le prestazioni di HashMap 438
Utility 439
Ordinamento e ricerche nelle List 444
Rendere immutabili una Collection o una Map 446
Sincronizzare una Collection o una Map 448
Meccanismo di fail-fast 449
Conservare i riferimenti 450
WeakHashMap 453
Contenitori di Java 1.0 e Java 1.1 455
Vector ed Enumeration 455
Hashtable 457
Stack 457
BitSet 459
Riepilogo 462

**Input/Output 465**

Classe File 466
Ottenere l'elenco di directory 466
Classi interne anonime 468
Utility per directory 471
Verificare e creare directory 478
Input e output 481
Tipi di InputStream 482
Tipi di OutputStream 484
Aggiunta di attributi e interfacce utili 485
Leggere da un InputStream con FilterInputStream 485
Scrivere su un OutputStream con FilterOutputStream 487
Classi Reader e Writer 488
Origini e destinazioni dei dati 489
Modificare il comportamento dei flussi 490
Classi non modificate 491
Classe RandomAccessFile 491
Utilizzi tipici dei flussi di I/O 492
Bufferizzazione dei file di input 492
Input dalla memoria 494
Formattazione dell'input dalla memoria 495
Nozioni fondamentali sui file di output 496
Variante abbreviata per l'output su file di testo 497
Memorizzazione e recupero dei dati 498

Lettura e scrittura di file ad accesso casuale 501
Flussi con pipe 503
Utility per la scrittura e la lettura dei file 503
Lettura di file binari 507
Standard I/O 508
Leggere da standard input 508
Convertire System.out in PrintWriter 509
Redirezione dello standard I/O 510
Controllo dei processi 511
Nuova libreria di I/O 514
Conversione dei dati 518
Ottenimento di tipi di dato primitivi 522
Buffer per la visualizzazione 524
Big endian e little endian 528
Manipolazione dei dati tramite buffer 529
I buffer in dettaglio 531
File mappati in memoria 535
Prestazioni 536
Blocco dei file 540
Blocco parziale di file mappati 541
Compressione 543
Compressione semplice con GZIP 544
Creazione di archivi di file con Zip 546
Archivi Java 548
Serializzazione di oggetti 551
Recupero di una classe 556
Controllare la serializzazione 557
Parola chiave transient 563
Un'alternativa all'interfaccia Externalizable 565
Gestione delle versioni 568
Utilizzo della persistenza 569
XML 577
API Preferences 581
Riepilogo 583

**Tipi enumerativi 585**

Funzionalità di base di enum 585
Utilizzo delle importazioni static con le enum 587
Aggiunta di metodi a un'enum 588
Sovrascrittura dei metodi di enum 590
Enum nelle dichiarazioni switch 591
Il mistero di values() 592
Implementazione, non ereditarietà 596
Selezione casuale 597
Utilizzo delle interfacce per l'organizzazione 598
Utilizzo di EnumSet in alternativa ai flag 605

Utilizzo di EnumMap 608
Metodi specifici per le costanti 609
Catena di responsabilità con le enum 614
Macchine a stati con enum 620
Multiple dispatching 628
Dispatching con enum 631
Utilizzo dei metodi specifici per le costanti 634
Dispatching con le EnumMap 637
Utilizzo di un array bidimensionale 638
Riepilogo 639

**Annotazioni 641**

Sintassi di base 643
Definizione delle annotazioni 643
Meta-annotazioni 645
Scrivere elaboratori di annotazioni 646
Elementi dell'annotazione 648
Limiti dei valori predefiniti 648
Generare file esterni 649
Soluzioni alternative 653
Le annotazioni non supportano l'ereditarietà 654
Implementazione dell'elaboratore 654
Utilizzo di apt per l'elaborazione delle annotazioni 658
Utilizzo del modello Visitor con apt 664
Test unitari basati sulle annotazioni 668
Utilizzo di @Unit con i generici 681
Le suite non sono necessarie 683
Implementazione di @Unit 683
Rimozione del codice di test 693
Riepilogo 696

**Supplementi 697**

Supplementi scaricabili 697
Thinking in C: i fondamenti di Java 698
Seminari Thinking in Java 698
Seminario Hands-On Java su CD 698
Seminario Thinking in Objects 699
Thinking in Enterprise Java 699
Thinking in Patterns (with Java) 700
Seminario Thinking in Patterns 700
Consulenza e revisione di progetti 701

**Risorse 703**

Software 703

Editor e ambienti IDE 704
Libri 704
Analisi e progettazione 705
Python 708
Bibliografia dell'autore 709

**Indice analitico** **711**

# Prefazione

L'autore di questo manuale si è avvicinato al linguaggio Java ritenendolo "semplicemente un altro linguaggio di programmazione" come in effetti è, sotto molti punti di vista.

Tuttavia, studiando Java in modo più approfondito, ha avuto modo di notare come lo scopo fondamentale di questo linguaggio fosse profondamente diverso da quello degli altri linguaggi conosciuti fino a quel momento.

Programmare significa gestire la complessità: quella del problema da risolvere, cui si aggiunge la complessità del sistema sul quale il problema viene risolto. Per queste ragioni, la maggior parte dei progetti di programmazione non giunge a buon fine. Tra l'altro, di tutti i linguaggi noti all'autore, ben pochi hanno affrontato questo aspetto della programmazione, ritenendo che l'obiettivo principale consistesse nel ridurre drasticamente la complessità dello sviluppo e la manutenzione del software.[1]

Ovviamente molte scelte progettuali nei diversi linguaggi hanno dovuto prendere atto di tale complessità, sebbene, a un certo punto, vi fossero altri elementi essenziali di cui tenere conto: gli stessi elementi che, alla fine, fanno sì che molti programmatori si ritrovino a "sbattere la testa contro il muro". Per esempio, C++ ha dovuto conservare la retrocompatibilità con il C, in modo da garantire una migrazione indolore per i programmatori C e nel contempo mantenere un'efficienza elevata.

Si tratta senza dubbio di considerazioni che hanno contribuito in modo notevole al successo di C++, originando però una maggiore complessità, tale da impedire il completamento di molti progetti. È certamente possibile attribuire la responsabilità ai programmatori e alla direzione aziendale, tuttavia perché non avvantaggiarsi di un particolare linguaggio, se consente di evitare errori di programmazione?

Un altro esempio è fornito da Visual BASIC (VB): legato al BASIC, VB non è stato realmente progettato per essere un linguaggio estensibile, pertanto tutte le estensioni accumulatesi con il tempo hanno prodotto una sintassi in alcuni casi davvero ingestibile.

---

1. L'autore ritiene, tuttavia, che il linguaggio Python sia quello che più si avvicina a questo obiettivo: si veda *www.python.org*.

Pur essendo retrocompatibile con *awk*, *sed*, *grep* e altri strumenti Unix per sostituire i quali è stato ideato, Perl viene spesso tacciato di produrre codice cosiddetto "*write-only*", difficilmente leggibile e interpretabile a distanza di tempo.

Di contro, nella progettazione di C++, VB, Perl e altri come SmallTalk, si è considerato almeno in parte il problema della complessità, con il risultato che questi linguaggi si sono dimostrati notevolmente efficaci per risolvere problemi specifici.

Ciò che ha colpito particolarmente l'autore è stato comprendere che, tra gli obiettivi progettuali di Sun, vi sia stato quello di limitare la complessità per il programmatore, quasi a voler affermare: "ci siamo preoccupati di ridurre i tempi e le difficoltà connesse con la produzione di codice affidabile". All'inizio questo ha portato a un codice la cui esecuzione non era particolarmente rapida (problema che si è poi ridimensionato), ma ha anche consentito una considerevole riduzione nei tempi di sviluppo: anche meno della metà del tempo richiesto per creare un analogo programma in C++.

Se questo consente di risparmiare tempo e denaro, Java non si ferma qui. Incorpora numerosi processi completi e importanti (tra cui il multithreading e la programmazione di rete) nel linguaggio e nelle librerie che a volte possono semplificarli. Infine, Java consente di affrontare problemi notevolmente articolati, quali i programmi multipiattaforma, le modifiche dinamiche del codice e la sicurezza, ognuno dei quali occupa, nella "scala di complessità" di ogni programmatore, una posizione variabile da *impedimento* a *ostacolo insormontabile*. Pertanto, nonostante i problemi di prestazioni, cui si è accennato, le promesse di Java sono considerevoli, poiché può trasformarci in programmatori assai più produttivi.

Comunque sia, Java aumenta la possibilità di comunicazione tra persone: ciò risulta evidente nella realizzazione di programmi, nel lavoro di gruppo, nella produzione di interfacce per comunicare con l'utente, nell'esecuzione di programmi su diversi sistemi e nella scrittura di programmi per comunicare via Internet.

I risultati della rivoluzione nella comunicazione potrebbero talvolta confondersi con gli effetti della movimentazione di grandi quantità di dati. La vera rivoluzione si riconoscerà, invece, perché saremo in grado di comunicare più facilmente: l'uno con l'altro, ma anche in gruppi e globalmente. Alcuni sostengono che tale rivoluzione consista nella formazione di una *mente globale*, risultante da un insieme sufficiente di persone e dotata di sufficienti interconnessioni. Java potrà anche non essere lo strumento che darà il via a una simile rivoluzione, in ogni caso l'esistenza di questa possibilità ha motivato l'autore nell'insegnare questo linguaggio.

# Java SE5 e SE6

Questa edizione del manuale tiene conto dei numerosi miglioramenti apportati al linguaggio Java, in ciò che Sun originariamente ha chiamato JDK 1.5, più tardi diventato JDK5 o J2SE5, e che infine ha perso il "2", ormai sorpassato, per trasformarsi in Java SE5. Molte modifiche apportate a Java SE5 sono state ideate per migliorare l'attività del programmatore. Come vedrete, i progettisti di Java non sono riusciti a soddisfare appieno questa esigenza, per quanto in generale abbiano compiuto notevoli passi nella direzione giusta.

Uno degli obiettivi principali di questa edizione consiste nel far comprendere tutti i miglioramenti presenti in Java SE5/6, esaminandoli e utilizzandoli nel prosieguo del volume. Ciò implica che questo manuale si assume l'onere, in un certo senso audace, di essere "esclusivo per Java SE5/6", e che quindi la gran parte del codice non sarà compilabile con le precedenti versioni. Nella fase di build, infatti, Java visualizzerà errori e si interromperà.

Se per qualsiasi motivo siete vincolati a versioni anteriori, tenete presente che anche questa edizione fornisce le basi del linguaggio, e che le versioni precedenti di questo manuale sono comunque disponibili per il download gratuito da *www.mindview.net*. Per varie ragioni, si è deciso di non fornire gratuitamente la versione digitale dell'edizione corrente, ma soltanto quelle precedenti.

## *Java SE6*

Questo manuale è un progetto imponente che ha richiesto molto tempo. Prima della sua pubblicazione è stata presentata la versione beta di Java SE6 (nome in codice *mustang*). Nonostante la presenza di modifiche secondarie, che hanno consentito di perfezionare alcuni degli esempi presenti nel libro, in genere le nuove caratteristiche di Java SE6 non hanno influito sui contenuti del volume; le novità riguardano soprattutto la velocità di esecuzione e le funzionalità di libreria che non rientrano negli scopi di questo manuale.

Il codice esposto in questo libro è stato testato con successo con una versione *release-candidate* di Java SE6, che non dovrebbe presentare variazioni tali da influire sul contenuto del manuale; qualora la versione definitiva di Java SE6 dovesse essere caratterizzata da modifiche sostanziali, esse saranno riportate nel codice sorgente del manuale, liberamente scaricabile da *www.mindview.net*.

Questo manuale è destinato a Java SE5/6, il che significa che è stato scritto tenendo in considerazione Java SE5 e le modifiche significative introdotte da questa versione, pur essendo ugualmente applicabile anche a Java SE6.

## La quarta edizione

La soddisfazione di realizzare la nuova edizione di un manuale consiste nel poter fare tesoro dell'esperienza acquisita con la precedente, e nell'opportunità di rivedere passaggi difficili o semplicemente noiosi. Come sempre la preparazione di una nuova edizione dà origine a idee affascinanti e inedite, e l'imbarazzo che potrebbe sorgere dalla rilettura del proprio lavoro è attenuato dal piacere di esprimere le proprie idee in una forma sempre migliore.

Ma nella mente dell'autore vi è anche la sfida a produrre un lavoro che perfino i possessori delle edizioni precedenti desiderino acquistare: ciò lo stimola a migliorare, riscrivere e riorganizzare quanto più possibile, per offrire ai lettori una nuova esperienza costruttiva.

### *Modifiche*

Il CD-ROM che in passato veniva fornito come parte integrante del manuale non è incluso in questa edizione. La parte più importante del CD, il seminario multimediale *Thinking in C*, prodotto per MindView da Chuck Allison, ora è disponibile come presentazione in formato Flash, e scaricabile liberamente in lingua inglese. L'obiettivo di questo seminario è fornire ai lettori che non avessero familiarità con la sintassi del C le competenze indispensabili per comprendere il materiale presentato in questo manuale. Nonostante due capitoli siano dedicati a un'introduzione alla sintassi del linguaggio C, questi potrebbero rivelarsi insufficienti per chi non disponesse della necessaria esperienza: *Thinking in C* è stato progettato espressamente per aiutare questi utenti a conseguire il livello di competenza necessario.

Pur essendo stato interamente riscritto tenendo conto delle principali modifiche intervenute nelle librerie Java SE5 "Concurrency Utilities", il Capitolo 1 del Volume 3, *La concorrenza* (*Multithreading*, nelle edizioni precedenti in lingua inglese), fornisce ancora i fondamenti teorici della programmazione concorrente. Senza queste basi sarebbe arduo comprendere gli argomenti più complessi relativi al threading. L'autore ha trascorso diversi mesi lavorando su queste caratteristiche di Java, in quel mondo sotterraneo chiamato "programmazione concorrente": ne è risultato un capitolo che, oltre ai fondamenti teorici, si addentra in una trattazione più avanzata.

È presente un nuovo capitolo per ogni nuova caratteristica importante introdotta in Java SE5, mentre altre novità sono state incorporate nel materiale esistente. Grazie all'impegno continuativo dell'autore nello studio dei design pattern, il manuale è stato arricchito di nuovi pattern.

L'autore ha sottoposto il manuale a una profonda riorganizzazione, soprattutto grazie all'esperienza maturata con l'insegnamento e alla consapevolezza che il concetto di "capitolo" meritasse qualche ripensamento. In generale, si è considerato che per meritare un intero capitolo l'argomento dovesse essere sufficientemente corposo. Tuttavia, in particolare nelle lezioni sul design pattern, l'autore ha notato che i partecipanti ai seminari ottengono risultati migliori se si presenta una struttura per volta, immediatamente seguita da un'esercitazione; questo approccio è anche più gratificante per il docente. Per tali motivi in questa versione del manuale i capitoli sono stati suddivisi per argomento, senza tenere conto della loro dimensione. L'autore è convinto che ciò rappresenti un miglioramento.

Grande importanza è stata assegnata anche alla verifica del codice. Senza una struttura di test incorporata che esegue verifiche ogniqualvolta si procede alla compilazione di un progetto, non si può in alcun modo valutare l'affidabilità del codice. A questo scopo è stata creata una struttura di test, scritta in linguaggio Python, per visualizzare e convalidare l'output di ogni programma: potete scaricarla dal sito *www.mindview.net*, nel codice relativo a questo manuale. L'argomento dei test in generale è trattato nel supplemento che potrete consultare all'indirizzo *http://mindview.net/Books/BetterJava*.

Sono stati riesaminati anche tutti gli esempi del manuale, con l'obiettivo di rispondere alla domanda: "per quale ragione è stata scelta questa soluzione?". Nella maggior parte dei casi sono state apportate modifiche e migliorie, con il duplice intento di rendere gli esempi più coerenti e fornire migliori "best practice" per la programmazione in Java, seppure nei limiti che caratterizzano un testo introduttivo. Molti esempi sono stati significativamente riprogettati e reimplementati; altri di scarsa rilevanza sono stati rimossi e ne sono stati aggiunti di nuovi.

I lettori hanno espresso commenti lusinghieri sulle prime tre edizioni di questo manuale, e ciò è stato indubbiamente gratificante per l'autore. Tuttavia vi sono e vi saranno anche alcune critiche: una delle più ricorrenti è relativa alle dimensioni del volume. In proposito, si potrebbe citare il famoso commento dell'Imperatore d'Austria quando, ascoltando un brano suonato da Mozart, esclamò: "Troppe note!". L'autore non ha alcuna pretesa di paragonarsi a Mozart, beninteso, tuttavia è ipotizzabile che chi esprime questo tipo di commento non abbia piena consapevolezza della vastità del linguaggio Java, o che ancora non abbia visto altri manuali sull'argomento.

In ogni caso, in questa edizione si è cercato di eliminare le parti obsolete o quantomeno non essenziali. Di norma è stato controllato tutto, eliminato quanto non era più necessario, apportate le modifiche e si è tentato di migliorare ove possibile. Il materiale eliminato originale è sempre disponibile per il

download dal sito *www.mindview.net*, nelle prime tre edizioni, unitamente ai supplementi a questa edizione.

Ai lettori che ancora non "digeriscono" la corposità di questo manuale vanno le scuse dell'autore, che ha comunque lavorato duramente per contenerne le dimensioni.

## Note sulla grafica di copertina

La copertina di *Thinking in Java* si ispira all'American Arts & Crafts Movement, iniziato alla fine del 1800 e giunto al suo apice tra il 1900 e il 1920. Questo movimento ha avuto origine in Inghilterra come reazione sia alla produzione meccanizzata della Rivoluzione industriale, sia allo stile molto ornamentale dell'epopea Vittoriana. L'Arts & Crafts evidenzia l'essenzialità della progettazione, le forme della natura ereditate dall'Art Nouveau, nonché la manualità e perizia del disegnatore, che ancora non disponeva di strumenti moderni.

Vi sono molte analogie con la situazione odierna: il cambio di secolo, l'evoluzione dalle rozze origini della rivoluzione informatica a qualcosa di più raffinato e ricco di significato, e l'enfasi sulla padronanza del software invece della semplice produzione del codice.

L'autore considera Java nella stessa ottica: un tentativo di elevare il programmatore da semplice meccanico del sistema operativo, per trasformarlo in un artigiano-artista del software.

Sia l'autore sia il disegnatore della copertina, amici fin dall'infanzia, trovano ispirazione in questo movimento, ed entrambi possiedono mobili, lampade e accessori originali o ispirati a questo periodo storico.

L'altro tema della copertina si ispira a un contenitore per la raccolta di insetti, analogo a quelli usati dai naturalisti per preservare i propri esemplari. Questi insetti sono *oggetti* inseriti negli *oggetti-contenitore*, che a loro volta si trovano all'interno dell'*oggetto-copertina*: una metafora che illustra bene il concetto fondamentale di aggregazione nella programmazione a oggetti. Naturalmente un programmatore non potrà esimersi dall'associare gli insetti al termine *bug* (insetto, appunto): i bug che sono stati catturati, presumibilmente resi innocui in un contenitore e infine confinati in un piccolo espositore, quasi a sottolineare la capacità di Java di trovare, visualizzare e neutralizzare i bug. Del resto, questa è effettivamente una delle caratteristiche più potenti del linguaggio.

In questa edizione l'autore ha dipinto anche l'acquerello che è servito da sfondo per l'illustrazione di copertina.

## Ringraziamenti dell'autore

Per prima cosa desidero ringraziare i soci che mi hanno supportato durante i seminari, mi hanno fornito consulenza tecnica e mi hanno aiutato nello sviluppo dei progetti di insegnamento: Dave Bartlett, Bill Venners, Chuck Allison, Jeremy Meyer e Jamie King. Apprezzo sempre più la vostra pazienza, e apprezzo che personalità indipendenti come le nostre possano collaborare in modo ottimale.

Di recente, senza dubbio grazie a Internet, un gran numero di persone si è associato alle mie imprese, di norma lavorando dai propri uffici. Per collaborare su scala così vasta in passato sarebbe stato necessario prendere in affitto un ambiente piuttosto grande, ma grazie a Internet, alla FedEx e al telefono, mi è possibile usufruire del loro contributo senza costi aggiuntivi.

Nei miei tentativi di imparare a "giocare bene con gli altri", queste persone mi sono state molto utili, e confido di avere l'opportunità di continuare a migliorare il mio lavoro traendo beneficio dagli sforzi altrui. Paula Steuer mi ha fornito un supporto inestimabile mettendo ordine nella gestione della mia attività, e mantenendolo: grazie per avermi stimolato quando era necessario, Paula. Jonathan Wilcox ha passato al setaccio la mia struttura aziendale, rivoltando ogni sasso che potesse nascondere scorpioni o altre insidie, e guidandomi in tutti gli aspetti legali.

Grazie per la vostra attenzione e la vostra tenacia. Sharlynn Cobaugh è ormai diventata un'esperta nell'elaborazione audio e un elemento essenziale nella realizzazione dei corsi multimediali, come pure nella soluzione di altri problemi. Grazie per la perseveranza dimostrata in occasione di problemi informatici all'apparenza insolubili. I collaboratori di Amaio a Praga mi sono stati di aiuto in numerosi progetti. Daniel Will-Harris mi ha ispirato l'idea di lavorare via Internet e naturalmente è stato fondamentale per tutte le progettazioni grafiche. Nel corso degli anni, attraverso conferenze e workshop, Gerald Weinberg è ufficiosamente diventato mio allenatore e mentore, e per questo lo ringrazio.

Ervin Varga si è rivelato straordinariamente efficace per le sue correzioni tecniche sulla quarta edizione: benché altri abbiano fornito aiuto su diversi capitoli ed esempi, Ervin è stato il revisore tecnico principale del manuale, incaricandosi anche di riscrivere la guida alle soluzioni della quarta edizione. Ervin ha corretto molti errori e operato miglioramenti che si sono rivelati inestimabili aggiunte a questo testo. La sua accuratezza e attenzione per i dettagli sono sorprendenti: un grazie a Ervin, certamente il migliore lettore che abbia mai avuto.

Il blog che scrivo sul sito *www.artima.com* di Bill Venners è stato utilissimo per verificare la validità delle mie idee. Ringrazio i lettori che mi hanno aiutato a chiarire molti concetti esprimendo i loro commenti: James Watson, Howard Lovatt, Michael Barker e altri, in particolare quanti mi hanno aiutato con i tipi generici.

Grazie a Mark Gallesi per la sua ininterrotta assistenza.

Evan Cofsky continua a essere un valido sostenitore, grazie alla sua competenza sugli oscuri dettagli di configurazione e gestione dei server web Linux, e per avere ben configurato e reso sicuro il server di MindView.

Un ringraziamento speciale al mio nuovo amico, il caffè, che ha contribuito a generare un immenso entusiasmo per questo progetto. Il bar Camp4 Coffee di Crested Butte (Colorado) è diventato il rifugio principale dei partecipanti ai seminari di MindView, e durante le pause si è dimostrato il miglior fornitore di catering. Un ringraziamento va all'amico Al Smith per averlo creato e reso un luogo così confortevole, e per essere una persona così interessante e divertente. Grazie anche a tutti i baristi del Camp4 Coffee, sempre di ottimo umore.

Desidero ringraziare i collaboratori di Prentice Hall che continuano a soddisfare le mie esigenze, anche quelle più particolari, e per fare del loro meglio per semplificarmi il lavoro.

Alcuni strumenti mi sono stati particolarmente utili durante lo sviluppo, e per questa ragione voglio ringraziare i loro creatori. Cygwin (*www.cygwin.com*) ha risolto innumerevoli problemi che Windows non poteva o non voleva risolvere: se soltanto avessi potuto servirmene 15 anni fa, quando ancora utilizzavo Gnu Emacs! Eclipse di IBM (*www.eclipse.org*) rappresenta un contributo straordinario alla comunità di sviluppo: mi aspetto grandi cose da questa piattaforma, il cui sviluppo procede ancora oggi. IntelliJ Idea di JetBrains (*www.jetbrains.com*) continua a essere un riferimento creativo tra gli strumenti di sviluppo.

Con questo manuale ho iniziato a usare Enterprise Architect di Sparxsystems (*www.sparxsystems.com*), che è diventato rapidamente il mio strumento UML preferito. Il formattatore di codice Jalopy di Marco Hunsicker (*www.triemax.com*) mi è stato utile in numerose occasioni, e Marco si è rivelato impagabile nel configurare questo strumento secondo le mie esigenze. Anche JEdit di Slava Pestov (*www.jedit.org*) e la sua collezione di plug-in mi sono stati spesso preziosi: JEdit è un eccellente editor per principianti da utilizzare durante i seminari.

Naturalmente, e non lo dirò mai abbastanza, per risolvere i problemi mi servo del linguaggio Python (*www.python.org*), delle idee del mio amico Guido

Van Rossum e della banda di geni pazzoidi con i quali ho trascorso giorni memorabili: Tim Peters, ho incorniciato quel mouse che hai preso a prestito, ora ufficialmente chiamato il *TimBotMouse*. Una sola raccomandazione: cercate posti più sani dove pranzare. Grazie ovviamente all'intera comunità di Python, un gruppo di persone straordinarie.

Molte persone mi hanno fatto pervenire le loro correzioni e a tutte sono debitore, ma un grazie particolare (per la prima edizione) va a Kevin Raulerson (scopritore di tonnellate di bug), Bob Resendes (semplicemente incredibile), John Pinto, Joe Dante, Joe Netto (tutti e tre favolosi), David Combs (per i molti chiarimenti e le correzioni di grammatica), Dr. Robert Stephenson, John Cook, Franklin Chen, Zev Griner, David Karr, Leander A. Stroschein, Steve Clark, Charles A. Lee, Austin Maher, Dennis P. Roth, Roque Oliveira, Douglas Dunn, Dejan Ristic, Neil Galarneau, David B. Malkovsky, Steve Wilkinson e un nugolo di altre persone. Il professor Marc Meurrens si è impegnato a fondo per promuovere e realizzare la versione elettronica della prima edizione del manuale disponibile in Europa.

Grazie a quanti mi hanno aiutato a riscrivere gli esempi di utilizzo della libreria Swing (per la seconda edizione) e che mi hanno fornito assistenza: Jon Shvarts, Thomas Kirsch, Rahim Adatia, Rajesh Jain, Ravi Manthena, Banu Rajamani, Jens Brandt, Nitin Shivaram, Malcolm Davis, e tutti coloro che mi hanno dato supporto.

Nella quarta edizione, Chris Grindstaff è stato molto utile nello sviluppo della sezione SWT e Sean Neville ha scritto la prima bozza della sezione Flex.

Kraig Brockschmidt e Gen Kiyooka sono stati tra i pochi brillanti tecnici di cui sono diventato amico: autorevoli e fuori dagli schemi, praticano yoga e altre forme di innalzamento spirituale che ritengo veramente ispiratrici ed educative.

Non mi ha sorpreso scoprire quanto la comprensione di Delphi mi sia stata utile per Java, dal momento che questi linguaggi condividono molti concetti e tecniche di progettazione. I miei amici esperti di Delphi hanno contribuito ad approfondire la mia conoscenza di questo meraviglioso ambiente di programmazione. In particolare Marco Cantu (un altro italiano: forse le origini latine comportano un'attitudine per linguaggi di programmazione?), Neil Rubenking (cultore di yoga, cucina vegetariana e Zen finché non ha scoperto i computer) e naturalmente Zack Urlocker (il product manager originale di Delphi), un amico di vecchia data con il quale ho girato il mondo.

Noi tutti siamo debitori dell'intelligenza vivace di Anders Hejlsberg, che continua a lavorare duramente su C# (che, come apprenderete in questo manuale, è stata la fonte principale di ispirazione per Java SE5).

Le intuizioni e il supporto del mio amico Richard Hale Shaw sono stati molto utili, come quelli di Kim, del resto. Richard e io abbiamo trascorso molti mesi tenendo insieme seminari e sforzandoci di fornire ai partecipanti l'esperienza di apprendimento perfetta.

La progettazione del manuale e della copertina, così come l'illustrazione della copertina stessa, sono state curate dal mio amico Daniel Will-Harris, noto autore e designer (*www.will-harris.com*), che era solito giocare con lettere adesive mentre aspettava l'invenzione dei computer e del desktop publishing, e che si lamentava di me borbottando sui miei problemi di algebra. In ogni caso, sappiate che io stesso ho prodotto le bozze finali, pertanto eventuali errori sono da imputare soltanto a me.

Ho usato Microsoft Word XP per Windows per scrivere il manuale e creare le pagine già pronte in Adobe Acrobat; il volume è stato realizzato direttamente dai file PDF Acrobat. Quando ho prodotto le versioni finali della prima e della seconda edizione del libro mi trovavo all'estero: come omaggio all'era elettronica, la prima edizione è stata inviata da Città del Capo (Sudafrica) mentre la seconda è stata trasmessa da Praga; la terza e la quarta edizione sono state preparate a Crested Butte, Colorado.

Un ringraziamento speciale va a tutti i miei insegnanti e a tutti i miei studenti (che sono anche i miei insegnanti). Molly, il gatto, si è accoccolato spesso sulle mie ginocchia mentre lavoravo, garantendomi il suo contributo caldo e peloso.

Tra gli amici che mi hanno sostenuto: Patty Gast, Andrew Binstock, Steve Sinofsky, JD Hildebrandt, Tom Keffer, Brian McElhinney, Brinkley Barr, Bill Gates del Midnight Engineering Magazine, Larry Constantine e Lucy Lockwood, Gene Wang, Dave Mayer, David Intersimone, Chris e Laura Strand, gli Almquists, Brad Jerbic, Marilyn Cvitanic, Mark Mabry, le famiglie Robbins, le famiglie Moelter e i McMillans, Michael Wilk, Dave Stoner, i Cranstons, Larry Fogg, Mike Sequeira, Gary Entsminger, Kevin e Sonda Donovan, Joe Lordi, Dave e Brenda Bartlett, Patti Gast, Blake, Annette & Jade, i Rentschlers, i Sudeks, Dick, Patty e Lee Eckel, Lynn e Todd e le loro famiglie.

Senza dimenticare, naturalmente, mamma e papà.

# Introduzione

*"E diede all'uomo la parola, e la parola generò il pensiero, che è la misura dell'Universo"*
*Percy Bysshe Shelley, Prometeo liberato (a. II sc. IV)*

*"Gli esseri umani... sono molto spesso alla mercé dello specifico linguaggio che è diventato il mezzo di espressione per la loro società. È illusorio ritenere di poter adattare la realtà in modo fondamentale senza l'uso della lingua, e che la lingua sia soltanto un mezzo fortuito per risolvere gli specifici problemi di comunicazione e riflessione. La questione è che, in larga misura, 'il mondo reale' si è inconsapevolmente sviluppato sulle consuetudini linguistiche del gruppo."*
*Edward Sapir, "La posizione della linguistica come scienza", in Cultura, linguaggio e personalità, Torino. Einaudi, 1972.*

Come qualsiasi linguaggio umano, Java rappresenta un modo per esprimere concetti. Se avrà successo, sarà certamente più semplice e flessibile dei linguaggi alternativi, via via che aumenteranno la dimensione e la complessità dei problemi da risolvere.

Non è possibile considerare Java come una banale raccolta di funzionalità: se considerate singolarmente, alcune di esse non hanno senso. Potete riferirvi alla somma delle parti soltanto pensando in termini di *progettazione*, non di semplice scrittura del codice. E per capire Java in questo modo è necessario comprendere i problemi relativi ai linguaggi e alla programmazione in generale. Questo manuale espone alcuni problemi di programmazione, spiega le ragioni per cui sono definiti tali ed espone le tecniche Java per la loro risoluzione. Quindi, l'insieme di caratteristiche presentato in ciascun capitolo si basa sul modo in cui un determinato tipo di problema viene risolto usando questo linguaggio. Attraverso questo approccio riuscirete, un po' alla volta, a considerare la *forma mentis* Java come vostra lingua madre.

Nel corso del manuale considereremo la necessità di costruire un modello mentale che vi consenta una profonda comprensione del linguaggio; questo vi consentirà, quando incontrerete un problema, di assimilarlo al vostro modello e dedurre la risposta.

## Prerequisiti

Questo manuale presuppone una certa pratica di programmazione: occorre sapere che un programma è un insieme di dichiarazioni; comprendere i concetti di *subroutine*, funzione, macro; conoscere le istruzioni di controllo del flusso programmatico, come *if*, i costrutti per la gestione dei cicli, come *while*, e così via. Potreste aver appreso queste nozioni lavorando in ambienti diversi, per esempio programmando con un macrolinguaggio oppure utilizzando uno strumento come Perl: in ogni caso, purché abbiate sufficiente esperienza da sentirvi a vostro agio con i concetti fondamentali della programmazione, potrete trarre vantaggio dalla lettura di questo volume.

Naturalmente sarà più facile seguire questo manuale per i programmatori C, e ancora di più per i programmatori C++. Non preoccupatevi, tuttavia, se non siete esperti in questi linguaggi: che lo siate o meno, quello che serve è tanta buona volontà e duro lavoro. Inoltre, il seminario multimediale *Thinking in C*, che potete scaricare da *www.mindview.net*, vi fornirà i fondamenti di programmazione necessari per apprendere Java. Comunque sia, vedrete i concetti della programmazione a oggetti (OOP) e i meccanismi di controllo di base Java.

Malgrado la presenza di rimandi a funzionalità tipiche di C e C++, questi non devono essere visti come considerazioni riservate agli eletti, bensì un supporto affinché come programmatori possiate mettere Java in relazione con questi linguaggi, dai quali, dopotutto, è derivato. L'autore ha avuto cura di semplificare al massimo questi riferimenti, chiarendo eventuali argomenti che possano risultare poco familiari a un programmatore non esperto di C/C++.

## Imparare Java

Nello stesso periodo in cui è stato pubblicato il suo primo libro, *Using C++* (Osborne/McGraw-Hill, 1989), l'autore ha iniziato a studiare Java. Insegnare idee e concetti di programmazione è poi diventata la sua professione: in tutto il mondo, fin dal 1987 l'autore ha visto persone assentire col capo, sguardi vuoti ed espressioni perplesse tra il pubblico.

Dopo aver iniziato l'istruzione aziendale di gruppi più piccoli, durante le esercitazioni l'autore si è reso conto che anche quanti sorridevano e chinavano il capo in segno di assenso erano poi confusi su molti argomenti. Organizzando e presiedendo per molti anni i corsi di C++ alla Software Development Conference, e in seguito quelli di Java, l'autore ha scoperto che

i relatori avevano la tendenza a fornire al pubblico troppi argomenti e più velocemente del dovuto.

Era evidente che in tal modo, sia per il diverso livello dei partecipanti sia per il modo in cui venivano presentati gli argomenti, si sarebbe finito per perdere l'attenzione di una parte del pubblico. Forse perché refrattario ai tradizionali metodi di insegnamento (avversione che per molte persone nasce dalla noia), l'autore ha ritenuto di dare una svolta realizzando presentazioni di tipo diverso, tutte piuttosto brevi, ricorrendo all'esperienza acquisita dalla sperimentazione e dalla ripetizione, una tecnica che peraltro funziona bene anche nella progettazione software.

Alla fine, utilizzando quanto appreso dalla propria esperienza di insegnamento, l'autore ha sviluppato un tipo di corso che la sua società, MindView Inc., offre come seminario intitolato *Thinking in Java*, tenuto sia internamente sia per il grande pubblico: è il principale corso introduttivo di MindView, atto a fornire le basi necessarie per corsi più avanzati. Sul sito *www.mindview.net* potete trovare maggiori informazioni su questo seminario, disponibile anche nel CD-ROM *Hands-On Java*.

Il feedback ottenuto da ogni corso permette di modificare e riorganizzare la materia, fino a trovare la tipologia di insegnamento più adatta. Questo libro, tuttavia, non è una semplice raccolta di annotazioni per un corso, ma cerca di fornire quante più informazioni possibile, strutturando i diversi temi per guidarvi all'argomento successivo. Più di ogni altra cosa, il manuale è stato concepito per aiutare il lettore ad affrontare in modo autonomo un nuovo linguaggio di programmazione.

## Obiettivi

Come il titolo precedente, *Thinking in C++*, questo libro è stato progettato con un obiettivo preciso: fornire gli elementi per apprendere un linguaggio di programmazione. Quando progetta un capitolo del manuale, l'autore tiene conto anche del risultato di una buona lezione tenuta nei suoi corsi. Il feedback del pubblico ha consentito di individuare i concetti più complessi, che richiedono maggiore approfondimento. Laddove l'ambizione aveva portato a esaminare un numero eccessivo di funzionalità, durante i corsi si è poi compreso che questo creava confusione negli studenti.

Gli obiettivi principali di questo manuale sono descritti di seguito.

1. Presentare il materiale un passo alla volta, per consentire di assimilare facilmente ogni concetto prima di passare a un nuovo argomento. La presentazione delle varie caratteristiche è stata ordinata accuratamente,

in modo che ciascun argomento sia esposto nella sua interezza prima di vederlo in azione. Naturalmente questo non è sempre possibile: in questi casi, è fornita una breve descrizione introduttiva.

2. Ricorrere a esempi quanto più semplici e brevi possibile. Questo approccio talvolta impedisce di affrontare problemi derivati dal mondo reale; tuttavia, il principiante trova più gratificante riuscire a capire ogni dettaglio di un esempio, che non essere impressionato dall'ampiezza del problema che viene risolto. Bisogna anche tenere conto di un serio limite alla quantità di codice che può essere assimilato in aula. Per questa ragione l'autore si aspetta di ricevere critiche per i suoi "esempi giocattolo", ma è pronto ad affrontarle per realizzare un prodotto utile anche dal punto di vista pedagogico.
3. Fornire le competenze importanti ai fini della comprensione del linguaggio, anziché tutto il nozionismo dell'autore. Esiste una gerarchia nell'importanza delle informazioni: vi sono alcune nozioni che il 95% dei programmatori non avrà occasione di usare, dettagli che confondono lo studente e aumentano la sua percezione della complessità del linguaggio. Considerate un semplice esempio preso dal linguaggio C nel quale, memorizzando la tavola delle precedenze degli operatori (operazione che l'autore non ha mai fatto), è possibile scrivere dell'ottimo codice. Se si tiene conto anche di questi argomenti, il semplice lettore o manutentore del codice risulterà confuso: è quindi preferibile ignorare la precedenza degli operatori e servirsi delle parentesi in caso di dubbi.
4. Mettere a fuoco ogni sezione, in modo che il tempo richiesto per la lettura e quello necessario per eseguire le esercitazioni siano contenuti. Questo non solo fa sì che la mente del pubblico sia più attenta e coinvolta durante un workshop, ma fornisce al lettore una più intensa sensazione di autorealizzazione.
5. Fornire una base solida che consenta di assimilare gli argomenti in modo adeguato prima di passare a corsi o manuali più approfonditi.

## Imparare da questo libro

L'edizione originale di questo manuale è stata sviluppata traendo spunto da un corso di una settimana: un periodo di tempo sufficiente, quando Java era nella sua infanzia, per illustrare le caratteristiche del linguaggio. Via via che Java cresceva e si arricchiva di funzionalità e librerie, l'autore ha tentato ostinatamente di contenere tutto in un periodo di tempo così breve, fino a quando un cliente non chiese che venissero spiegate "soltanto le basi". A

quel punto, l'autore si è reso conto che il tentativo di insegnare tutto in una settimana era diventato angosciante sia per il docente sia per i partecipanti: Java non era più il linguaggio "semplice" che potesse essere spiegato compiutamente in pochi giorni.

Questa esperienza ha governato gran parte della riorganizzazione di questo libro, ora progettato per supportare un seminario della durata di due settimane oppure due trimestri di corso universitario. La parte introduttiva termina con il Capitolo 1 del Volume 2, "Gestione degli errori con le eccezioni", e può essere integrata con un'introduzione a JDBC, Servlet e JSP: questa sezione rappresenta un corso di base, ed è il nucleo del CD-ROM *Hands-On Java*. Le altre parti del manuale formano un corso di livello intermedio, corrispondente al materiale presente nel CD-ROM *Intermediate Thinking in Java*. Entrambi i CD-ROM sono in vendita su *www.mindview.net*; per informazioni sui sussidi ai docenti relativi a questo manuale potete contattare l'editore Pearson Education Italia all'indirizzo *http://hpe.pearsoned.it*.

## Documentazione JDK in HTML

Il linguaggio e le librerie Java di Sun Microsystems, fornite gratuitamente all'indirizzo *http://java.sun.com*, vengono rese disponibili insieme con documentazione in forma elettronica, che potete consultare con qualsiasi browser web. Molti manuali su Java hanno duplicato questa documentazione: pertanto, dal momento che potreste già disporre di questa documentazione, e in ogni caso non avrete difficoltà a procurarvela in qualsiasi momento, questo manuale non ne ripeterà il contenuto. Del resto, non solo è più rapido trovare la descrizione di una classe cercandola con il browser web, rispetto alla consultazione di un libro, ma probabilmente la documentazione in linea è più aggiornata: troverete dunque alcuni rimandi alla "documentazione JDK". Vi saranno fornite descrizioni supplementari delle classi soltanto quando sia necessario integrare la documentazione per consentirvi di comprendere a fondo qualche specifico esempio.

## Esercizi

Si è notato che le esercitazioni sono incredibilmente utili al fine di perfezionare la comprensione di uno studente; per questo motivo, al termine di ogni capitolo vengono presentati alcuni esercizi. La maggior parte è ideata per essere svolta in un periodo di tempo ragionevole in aula sotto la supervisione di un docente, per assicurarsi che tutti gli studenti abbiano recepito corret-

tamente le nozioni fornite: alcuni esercizi sono più stimolanti, in ogni caso nessuno di essi rappresenta una sfida particolare.

Potete trovare le soluzioni di tali esercizi nel documento elettronico *The Thinking in Java Annotated Solution Guide*, in vendita su *www.mindview.net*.

## Fondamenti di Java

Un altro vantaggio di questa edizione è il seminario multimediale gratuito *Thinking in C*, scaricabile da *www.mindview.net*, che vi fornirà un' introduzione alla sintassi, agli operatori e alle funzioni del linguaggio C su cui si basa la sintassi Java. Nelle edizioni precedenti era presente nel CD-ROM *Foundations for Java* allegato al volume, ora potete scaricarlo gratuitamente.

La tecnologia si è evoluta, pertanto *Thinking in C* è stato rielaborato come presentazione Flash scaricabile da Internet. Mettendo a disposizione questo seminario in linea, invece che in un CD-ROM allegato, l'autore si assicura che ognuno possa partire con una preparazione adeguata.

Inoltre la presenza di questo seminario in linea consente l'accesso a questo libro da parte di un pubblico più vasto. Sebbene i Capitoli 3 e 4 del Volume 1, "Gli operatori" e "Il controllo dell'esecuzione" trattino gli elementi fondamentali di Java che discendono dal C, il corso in linea offre un'introduzione più semplice e libera dai presupposti sulle competenze di programmazione dello studente, a differenza del manuale.

## Codice sorgente

Tutto il codice sorgente presente in questo libro è *copyrighted freeware*, vale a dire disponibile gratuitamente ma soggetto a diritti d'autore, ottenibile come singolo pacchetto sul sito *www.mindview.net*. Per assicurarvi di disporre della versione più recente, ricordate che questo sito è il distributore ufficiale del codice. Siete liberi di utilizzare il codice in aula e in ambiente didattico.

L'obiettivo principale dei diritti d'autore è fare in modo che la fonte del codice venga segnalata correttamente e che tale codice non possa essere ripubblicato senza autorizzazione. Se la fonte del codice è citata, l'uso degli esempi nella maggior parte dei media è generalmente consentito. In ogni file sorgente troverete quindi un riferimento alla seguente informativa:

```
//:! Copyright.txt
This computer source code is Copyright ©2006 MindView, Inc.
```

All Rights Reserved.

Permission to use, copy, modify, and distribute this computer source code (Source Code) and its documentation without fee and without a written agreement for the purposes set forth below is hereby granted, provided that the above copyright notice, this paragraph and the following five numbered paragraphs appear in all copies.

1. Permission is granted to compile the Source Code and to include the compiled code, in executable format only, in personal and commercial software programs.

2. Permission is granted to use the Source Code without modification in classroom situations, including in presentation materials, provided that the book "Thinking in Java" is cited as the origin.

3. Permission to incorporate the Source Code into printed media may be obtained by contacting:

MindView, Inc. 5343 Valle Vista La Mesa, California 91941
Wayne@MindView.net

4. The Source Code and documentation are copyrighted by MindView, Inc. The Source code is provided without express or implied warranty of any kind, including any implied warranty of merchantability, fitness for a particular purpose or non-infringement. MindView, Inc. does not warrant that the operation of any program that includes the Source Code will be uninterrupted or error-free. MindView, Inc. makes no representation about the suitability of the Source Code or of any software that includes the Source Code for any purpose. The entire risk as to the quality and performance of any program that includes the Source Code is with the user of the Source Code. The user

```
understands that the Source Code was developed for research
and instructional purposes and is advised not to rely
exclusively for any reason on the Source Code or any
program that includes the Source Code. Should the Source
Code or any resulting software prove defective, the user
assumes the cost of all necessary servicing, repair, or
correction.

5. IN NO EVENT SHALL MINDVIEW, INC., OR ITS PUBLISHER BE
LIABLE TO ANY PARTY UNDER ANY LEGAL THEORY FOR DIRECT,
INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES,
INCLUDING LOST PROFITS, BUSINESS INTERRUPTION, LOSS OF
BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS, OR FOR
PERSONAL INJURIES, ARISING OUT OF THE USE OF THIS SOURCE
CODE AND ITS DOCUMENTATION, OR ARISING OUT OF THE INABILITY
TO USE ANY RESULTING PROGRAM, EVEN IF MINDVIEW, INC., OR
ITS PUBLISHER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE. MINDVIEW, INC. SPECIFICALLY DISCLAIMS ANY
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE. THE SOURCE CODE AND DOCUMENTATION PROVIDED
HEREUNDER IS ON AN "AS IS" BASIS, WITHOUT ANY ACCOMPANYING
SERVICES FROM MINDVIEW, INC., AND MINDVIEW, INC. HAS NO
OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES,
ENHANCEMENTS, OR MODIFICATIONS.

Please note that MindView, Inc. maintains a Web site which
is the sole distribution point for electronic copies of the
Source Code, http://www.MindView.net (and official mirror
sites), where it is freely available under the terms stated
above.

If you think you've found an error in the Source Code,
please submit a correction using the feedback system that you
will find at http://www.MindView.net.
///:~
```

Potete usare liberamente il codice nei vostri progetti, in aula e nelle vostre presentazioni, purché sia presente l'informativa sui diritti d'autore che appare in ogni file sorgente.

### Standard di codifica

Nel testo di questo manuale, gli identificatori (metodi, variabili e nomi di classe) sono evidenziati in **grassetto**. Anche la maggior parte delle parole chiave è riportata in grassetto, tranne quelle che sono usate così spesso da rendere il grassetto ridondante, per esempio "class".

Negli esempi di questo libro si è fatto ricorso a uno stile di codifica particolare, per quanto possibile conforme a quello utilizzato da Sun in quasi tutti i frammenti di codice che trovate sul suo sito web (*http://java.sun.com/docs/codeconv/index.html*): uno stile che sembra essere supportato dalla maggior parte degli ambienti di sviluppo Java.

Se avete letto nella versione originale altri manuali scritti dall'autore, avrete certamente notato che il suo stile di codifica coincide con quello di Sun. Questo è gratificante sebbene l'autore, per quanto a sua conoscenza, non abbia nulla a che fare con questa affinità.

Lo stile di formattazione è un argomento che potrebbe essere oggetto di ore di discussione; per questa ragione l'autore non ha la pretesa di imporre la propria come forma corretta, basandosi sugli esempi forniti: l'autore ha motivazioni personali per ricorrere a questo stile. Poiché Java è un linguaggio di programmazione nel quale lo stile non è vincolante, potete continuare a usare quello con cui vi trovate a vostro agio. Un modo per ovviare al problema della codifica consiste nell'uso di uno strumento come *Jalopy* (*www.triemax.com*), rivelatosi prezioso nello sviluppo di questo manuale, con cui modificare la formattazione nel modo che preferite.

I file sorgenti presenti nel libro sono stati testati con un sistema automatico e dovrebbero funzionare tutti senza errori di compilazione. Come si è detto, questo manuale tratta di Java SE5/6: qualora abbiate necessità di esaminare argomenti relativi alle altre versioni del linguaggio, ricordate che le precedenti edizioni sono liberamente scaricabili presso *www.mindview.net*.

## Errori

Per quanti accorgimenti uno scrittore adotti per rilevare gli errori, alcuni di essi si insinuano sempre nel suo lavoro, spesso particolarmente evidenti a un lettore "fresco". Se notate qualsiasi cosa che ritenete sia un errore tecnico

o del codice, utilizzate il link a questo manuale sul sito *www.mindview.net,* indicando l'errore e la correzione che intendete proporre; per segnalare refusi o errori di traduzione relativi alla traduzione italiana contattate Pearson Education Italia all'indirizzo di posta elettronica *editoriale@pearson.com.* La vostra collaborazione sarà apprezzata.

# Capitolo 1

# Gestione degli errori con le eccezioni

La filosofia di base di Java è che "il codice malfatto non sarà eseguito". Il momento ideale per intercettare un errore è durante la compilazione, ancora prima di eseguire il programma. Tuttavia non tutti i problemi possono essere rilevati in fase di compilazione e occorre gestire gli altri durante l'esecuzione, ricorrendo ad apposite convenzioni che permettono a chi ha generato l'errore di comunicare le informazioni adatte a un destinatario che saprà affrontare correttamente il problema.

La gestione potenziata degli errori è uno dei meccanismi più potenti di cui potete avvalervi per accrescere la robustezza del vostro codice. Il ripristino dagli errori è una preoccupazione fondamentale per ogni programma che scrivete, ma è ancora più importante in Java, un linguaggio che ha tra i suoi obiettivi primari quello di generare componenti utilizzabili da altri programmatori: affinché un sistema sia robusto devono esserlo tutti i suoi componenti. Fornendo un modello coerente per la segnalazione degli errori mediante le eccezioni, Java permette ai vari componenti di comunicare eventuali problemi al codice client, in modo assolutamente attendibile.

L'obiettivo della gestione delle eccezioni in Java è semplificare la creazione di programmi affidabili, anche di grandi dimensioni, utilizzando meno codice possibile, con la massima fiducia che la vostra applicazione non darà luogo a errori imprevisti. Apprendere la gestione delle eccezioni non è complesso, ed è una delle caratteristiche che forniranno benefici immediati e significativi al vostro progetto.

Poiché la gestione delle eccezioni è l'unico meccanismo ufficiale adottato da Java per la segnalazione degli errori, viene supportato direttamente dal compilatore: tenete presente, comunque, che questo manuale è ricco di esempi che possono essere realizzati anche senza tenere conto di questo aspetto. In questo capitolo vedrete come scrivere codice in grado di gestire correttamente le eccezioni che potrebbero presentarsi nei vostri metodi.

## Concetti fondamentali

I primi linguaggi di programmazione, come il C, spesso mettevano a disposizione vari schemi per la gestione delle eccezioni, che pur essendo generalmente supportati da convenzioni, non erano parte integrante del linguaggio. Di norma il programmatore restituiva un valore speciale o impostava un flag, e il destinatario era tenuto a controllare il valore o il flag restituiti per determinare se vi fosse una situazione anomala. Tuttavia, con il passare degli anni si è notato che i programmatori che si servono di una libreria hanno la tendenza a ritenersi invincibili: "Certo, gli errori possono capitare, ma non nel mio codice". Questo atteggiamento ha fatto sì che molti sviluppatori non verificassero la presenza di errori: d'altra parte, a volte questo controllo era davvero difficile da implementare.[1]

Se foste così coscienziosi da verificare sempre l'eventuale presenza di un errore dopo aver chiamato un metodo, il vostro codice rischierebbe di trasformarsi in un incubo illeggibile. Poiché i programmatori erano ancora soliti produrre applicazioni in questi linguaggi, erano riluttanti ad ammettere la verità: che un simile approccio alla gestione degli errori rappresentava un notevole vincolo alla creazione di programmi di grandi dimensioni, robusti e manutenibili.

La soluzione consiste nel riconoscere la natura casuale della gestione degli errori e nell'imporre convenzioni formali. Questa è una lunga storia, poiché le prime implementazioni della gestione delle eccezioni (*exception handling*) risalgono ai sistemi operativi degli anni '60, addirittura al leggendario "**on error goto**" di BASIC. Tuttavia, la gestione delle eccezioni di C++ affonda le sue radici su Ada, e Java è basato fondamentalmente su C++, con alcuni richiami a Object Pascal. In questo contesto il termine "eccezione" assume il significato di "obiezione posta nei confronti di qualcosa". Nel punto in cui si verifica il problema potreste non sapere come gestirlo, tuttavia sapete certamente che non potete continuare come se nulla fosse accaduto: dovete interrompervi e qualcuno, da qualche parte, deve sapere che cosa fare. Purtroppo non avete sufficienti informazioni sul

1. A titolo di esempio il lettore-programmatore C potrà controllare il valore restituito dalla comune funzione **printf()**.

contesto corrente per ovviare al problema, pertanto dovete delegarlo a un livello più elevato, dove qualcuno sia qualificato per prendere le decisioni opportune.

L'altro beneficio piuttosto importante delle eccezioni è che possono ridurre la complessità del codice per la gestione degli errori: senza di esse, infatti, dovete eseguire controlli alla ricerca di un errore e gestirlo in varie posizioni del programma. Con le eccezioni, di contro, non è necessario controllare se vi sono errori nel punto di chiamata a un metodo, poiché l'eccezione ne garantirà l'intercettazione. Avrete a che fare con il problema soltanto in un punto, il cosiddetto gestore delle eccezioni o *exception handler*: questo criterio consente di economizzare sulla scrittura del codice e mantiene segregato il codice che descrive ciò che avviene durante la normale esecuzione da quello che viene eseguito in caso di problemi. Di norma la lettura, la scrittura e il rilevamento degli errori nel codice risultano molto più semplici con le eccezioni che con la classica tecnica di gestione degli errori.

## Eccezioni di base

Una condizione eccezionale è un problema che impedisce il proseguimento del metodo o la continuazione dell'ambito corrente. È importante distinguere uno stato eccezionale da un normale problema, in cui il contesto fornisce abbastanza informazioni da consentire di risolvere in qualche modo la difficoltà: in una condizione eccezionale non è possibile continuare a garantire l'operatività poiché mancano le informazioni necessarie per gestire il problema nel contesto corrente. Non è possibile fare altro che uscire da questo contesto e confinare il problema in uno più elevato: questo è esattamente ciò che accade "sollevando" un'eccezione (*to throw an exception*).

Considerate il semplice esempio della divisione: se il vostro programma rischia di eseguire una divisione per zero è opportuno che controlliate questa condizione. Ma che cosa rappresenta un divisore a zero? È possibile che nel contesto del problema che state cercando di risolvere con il metodo corrente sappiate come gestire un divisore a zero. Ma se questo si presentasse come valore inatteso non sarete in grado di gestirlo, pertanto è opportuno che solleviate un'eccezione invece di continuare con l'esecuzione.

Quando producete un'eccezione si verificano alcuni eventi: in primo luogo viene generato l'oggetto di eccezione, allo stesso modo di qualsiasi altro oggetto Java: nell'heap, con **new**. Poi il percorso di esecuzione corrente, ossia quello che non è possibile proseguire correttamente, viene interrotto e il riferimento all'oggetto di eccezione viene espulso dal contesto corrente. A quel punto il meccanismo di gestione delle eccezioni assume il controllo e inizia a cercare una posizione adatta da cui continuare l'esecuzione: questa posizione

è il gestore delle eccezioni, che ha il compito di gestire il problema in modo che il programma possa modificare il suo comportamento o semplicemente continuare.

Per comprendere come viene sollevata un'eccezione, considerate il semplice esempio di un riferimento a un oggetto chiamato **t**. È possibile che sia stato passato un riferimento non inizializzato, pertanto è opportuno eseguire un controllo prima di chiamare un metodo utilizzando quel riferimento. Potete trasmettere le informazioni sull'errore a un altro contesto generando un oggetto che rappresenta le vostre informazioni e "sollevandolo" dal vostro contesto corrente. Questa operazione è appunto definita con l'espressione "sollevare un'eccezione":

```
if(t == null)
  throw new NullPointerException();
```

Il codice di questo esempio solleva l'eccezione nel contesto corrente, permettendovi di evitare di indagare ulteriormente sul problema, che come "per magia" verrà gestito altrove: vedrete tra poco esattamente dove.

Le eccezioni consentono di pensare a tutto ciò che fate in termini di transazioni, sulle quali vigilano le eccezioni: "...la premessa fondamentale delle transazioni è che i calcoli distribuiti richiedevano la gestione delle eccezioni. Le transazioni sono l'equivalente informatico di un contratto legale. Se qualche cosa non funziona, l'intero contratto è annullato."[2]

Potete anche considerare le eccezioni come una sorta di meccanismo di annullamento nativo, che mette a disposizione diversi punti di ripristino nel programma (da utilizzare comunque con una certa attenzione). Se una parte del programma si "guasta", l'eccezione "annulla" l'operazione per ritornare a un punto stabile e noto del codice.

Una delle funzioni più importanti svolte dalle eccezioni è la loro capacità di impedire al programma di proseguire nel percorso elaborativo, qualora si presentasse un imprevisto. Questo è stato un vero problema nei linguaggi come C e C++; in modo particolare per il C, che non prevedeva tecniche idonee per indurre un programma a interrompere forzatamente il flusso di esecuzione delle istruzioni: questa situazione ha determinato un lungo ritardo nell'affrontare i problemi. Le eccezioni permettono di costringere il programma ad arrestarsi per segnalarvi che cosa è accaduto, o (teoricamente) cosa fare per gestire il problema e ripristinare la normale condizione di stabilità.

2. Jim Gray, vincitore del premio Turing per il contributo dato dal suo gruppo alle transazioni, in un'intervista apparsa su *www.acmqueue.org*.

### Argomenti delle eccezioni

Come si è detto, al pari di qualsiasi oggetto Java, le eccezioni vengono sempre create nell'heap con la parola chiave **new**, che assegna la memoria necessaria e chiama un costruttore. Tutte le eccezioni standard possiedono due costruttori: il primo è quello predefinito, mentre il secondo accetta un argomento **String** in modo che possiate fornire le indicazioni pertinenti all'eccezione:

```
throw new NullPointerException("t = null");
```

Come vedrete, questa stringa può essere poi estratta ricorrendo a varie tecniche. La parola chiave **throw** fornisce risultati interessanti. Dopo la creazione dell'oggetto di eccezione con **new**, il riferimento risultante viene passato a **throw**. In effetti, l'oggetto viene "restituito" dal metodo, anche se normalmente il metodo non dovrebbe restituire quel tipo di oggetto. Un modo semplicistico di considerare la gestione delle eccezioni è assimilarla a una sorta di meccanismo di ritorno alternativo, sebbene questa analogia possa procurarvi qualche inconveniente qualora venga presa troppo "alla lettera". Potete anche uscire dai normali ambiti sollevando un'eccezione: in entrambi i casi Java restituirà un oggetto di eccezione e il metodo o l'ambito si interromperanno.

L'analogia con il meccanismo di ritorno standard termina qui, poiché il punto in cui avviene la restituzione è diverso rispetto alla normale chiamata di metodo: ritornerete infatti in un gestore di eccezioni appropriato, che nello stack delle chiamate potrebbe trovarsi a molti livelli di "distanza" rispetto al punto in cui è stata prodotta l'eccezione.

È anche possibile sollevare qualsiasi tipo di oggetto **Throwable**, che è la classe radice delle eccezioni; in effetti, di solito produrrete una diversa classe di eccezioni per ogni tipo di errore. Le informazioni sull'errore sono presenti sia all'interno dell'oggetto di eccezione sia implicitamente nel nome della classe di eccezioni, in modo che in un contesto superiore sia possibile determinare come gestire l'evento. Considerate, però, che spesso le uniche informazioni fornite sono il tipo di eccezione e l'assenza di ogni altro dato significativo nell'oggetto di eccezione.

## Come intercettare un'eccezione

Per esaminare come avviene l'intercettazione di un'eccezione dovete innanzitutto comprendere il concetto di *guarded region*, letteralmente "regione custodita, sorvegliata": si tratta della porzione di codice che potrebbe produrre eccezioni ed è seguita dal codice incaricato di gestirle.

## *Blocco try*

Se vi trovate all'interno di un metodo e sollevate un'eccezione, o se questa viene prodotta da un altro metodo chiamato da quello corrente, il metodo terminerà le sue operazioni. Per evitare che **throw** provochi l'uscita dal metodo potete impostare al suo interno un blocco di codice speciale, incaricato di intercettare l'eccezione; questa serie di istruzioni è denominata *blocco try* poiché è in questo punto che si possono "provare" (*to try*, appunto) le diverse chiamate di metodo. Il blocco **try** è un ambito normale preceduto dalla parola chiave **try**:

```
try {
  // Codice che potrebbe generare eccezioni
}
```

Se aveste dovuto eseguire lo stesso controllo accurato con un linguaggio di programmazione che non supporta la gestione delle eccezioni, avreste dovuto incorporare ogni chiamata di metodo nel codice di setup e di controllo degli errori, anche se lo stesso metodo venisse chiamato più volte. Grazie alla gestione delle eccezioni, l'intera funzionalità può essere inglobata in un blocco **try**, per fare in modo che l'intercettazione di tutte le eccezioni avvenga in un solo punto: questo significa ottenere codice più facile da scrivere e leggere, dal momento che l'obiettivo primario del codice non si confonde con la verifica degli errori.

## *Gestori di eccezioni*

Naturalmente l'eccezione prodotta deve finire da qualche parte: questo "posto" è il gestore delle eccezioni (*exception handler*), uno per ogni tipo di eccezione che desiderate intercettare. Questi gestori seguono immediatamente il blocco **try** e sono contrassegnati dalla parola chiave **catch**:

```
try {
  // Codice che potrebbe generare eccezioni
} catch(Type1 id1) {
  // Gestisce le eccezioni per il Type1
} catch(Type2 id2) {
  // Gestisce le eccezioni per il Type2
} catch(Type3 id3) {
  // Gestisce le eccezioni per il Type3
}

// ecc.
```

Ogni istruzione **catch** è una specie di minimetodo che accetta un solo argomento di un tipo particolare. L'identificativo (**id1**, **id2** ecc.) può essere utilizzato all'interno del gestore, proprio come se si trattasse dell'argomento di un metodo. Talvolta potreste non avere bisogno di ricorrere all'identificativo poiché il tipo di eccezione fornisce informazioni sufficienti per gestire l'eccezione, in ogni caso l'identificativo deve essere presente.

I gestori devono comparire subito dopo il blocco **try**. Quando l'elaborazione solleva un'eccezione, il codice cerca il primo gestore con un argomento equivalente al tipo di eccezione: a quel punto il flusso passa nell'istruzione **catch** corrispondente e l'eccezione è considerata gestita. La ricerca dei gestori termina una volta che l'istruzione **catch** è finita. A differenza dell'istruzione **switch**, in cui è necessario prevedere un **break** dopo ogni **case** per evitare l'esecuzione di quelli successivi, il meccanismo di gestione delle eccezioni esegue soltanto l'istruzione **catch** corrispondente.

Ricordate che, sebbene all'interno del blocco **try** la stessa eccezione possa essere generata da un certo numero di chiamate diverse, è comunque necessario ricorrere a un solo gestore.

## Confronto tra interruzione e ripresa

La teoria di gestione delle eccezioni prevede due modelli di base. Java supporta la cosiddetta *termination* (interruzione), adottata anche dalla maggior parte dei linguaggi, quali C++, C #, Python, D ecc.: secondo questo modello si suppone che l'errore sia così grave da non potere riprendere l'esecuzione del programma dal punto in cui si è verificata l'eccezione; in altri termini, chi ha previsto l'eccezione ha ritenuto che non vi fosse alcuna possibilità di gestire la situazione e volutamente ha interrotto l'esecuzione del programma.

L'alternativa è denominata *resumption* (ripristino): in questo caso il gestore delle eccezioni è tenuto a fare del proprio meglio per modificare la situazione, e il metodo che ha generato l'errore viene nuovamente elaborato, nell'ipotesi che un secondo tentativo possa avere successo. Implementare un ripristino significa confidare di continuare l'esecuzione dopo avere gestito l'eccezione.

Per fare in modo che Java adotti questo comportamento non dovete sollevare un'eccezione quando incontrate l'errore, bensì eseguire una chiamata a un metodo che risolva il problema. In alternativa, potete includere il blocco **try** in un ciclo **while** che continui a mantenere il flusso elaborativo all'interno del blocco **try**, finché il risultato non sia soddisfacente.

Storicamente, i programmatori che lavorano su sistemi operativi che supportano la gestione delle eccezioni mediante ripristino finiscono per adottare l'approccio di interruzione e ignorano la possibilità di recupero. Nella pra-

tica la tecnica di ripristino è poco utile, sebbene possa a una prima analisi sembrare conveniente. La ragione di questo è forse da ricercarsi nelle implicazioni insite nell'utilizzo della tecnica di ripristino. Un gestore orientato al ripristino richiederebbe di essere informato sull'eccezione prodotta e dovrebbe contenere codice specifico per la posizione del codice in cui è stata prodotta l'eccezione; questa esigenza rende il codice di gestione difficile da scrivere e da manutenere, soprattutto nelle grandi applicazioni (nelle quali l'eccezione può essere generata in molte situazioni).

## Generazione di eccezioni personalizzate

Non siete limitati all'utilizzo delle eccezioni disponibili in Java: la gerarchia di eccezioni Java, infatti, non può prevedere tutti gli errori che potrebbero essere segnalati e offre quindi la possibilità di aggiungere gestori di errore personalizzati relativi a un problema specifico che il codice potrebbe incontrare.

Per generare la vostra classe di eccezione personalizzata dovete ereditare da una classe di eccezioni, di preferenza una nella quale il significato degli errori sia il più vicino possibile a quelli che avete previsto. La tecnica più semplice per creare un nuovo tipo di eccezione è fare in modo che sia il compilatore a generare il costruttore predefinito; questo consente di ridurre sensibilmente la quantità di codice da scrivere.

```
//: exceptions/InheritingExceptions.java
// Creazione di eccezioni personalizzate.

class SimpleException extends Exception {}

public class InheritingExceptions {
  public void f() throws SimpleException {
    System.out.println("Throw SimpleException from f()");
    throw new SimpleException();
  }
  public static void main(String[] args) {
    InheritingExceptions sed = new InheritingExceptions();
    try {
      sed.f();
    } catch(SimpleException e) {
      System.out.println("Caught it!");
```

```
    }
  }
} /* Output:
Throw SimpleException from f()
Caught it!
*///:~
```

Il compilatore genera un costruttore predefinito, che automaticamente e in modo invisibile chiama il costruttore predefinito della classe di base. Naturalmente in questo caso non otterrete un costruttore **SimpleException(String)**, ma in pratica questo costruttore non è molto utilizzato. Come vedrete, l'elemento più importante da tenere in considerazione per un'eccezione è il nome della classe, quindi un'eccezione come quella mostrata è quasi sempre soddisfacente.

Nell'esempio precedente il risultato viene visualizzato a console, dove è automaticamente gestito ed esaminato tramite il meccanismo di visualizzazione dell'output utilizzato in questo libro. Tuttavia potreste voler inviare l'output d'errore al flusso di *standard error*, scrivendo il codice che registra su **System.err**. Questo è solitamente un modo migliore per trasmettere le informazioni di errore rispetto allo standard **System.out**; quest'ultimo infatti può essere ridiretto. Inviando l'output a **System.err**, invece, esso non verrà ridirezionato, pertanto l'utente avrà maggiori probabilità di notarlo.

Potete anche produrre una classe di eccezioni che possiede un costruttore che accetta un argomento **String**.

```
//: exceptions/FullConstructors.java

class MyException extends Exception {
  public MyException() {}
  public MyException(String msg) { super(msg); }
}

public class FullConstructors {
  public static void f() throws MyException {
    System.out.println("Throwing MyException from f()");
    throw new MyException();
  }
  public static void g() throws MyException {
    System.out.println("Throwing MyException from g()");
    throw new MyException("Originated in g()");
```

```
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MyException e) {
      e.printStackTrace(System.out);
    }
    try {
      g();
    } catch(MyException e) {
      e.printStackTrace(System.out);
    }
  }
} /* Output:
Throwing MyException from f()
MyException
        at FullConstructors.f(FullConstructors.java:11)
        at FullConstructors.main(FullConstructors.java:19)
Throwing MyException from g()
MyException: Originated in g()
        at FullConstructors.g(FullConstructors.java:15)
        at FullConstructors.main(FullConstructors.java:24)
*///:~
```

Il codice aggiunto è ridotto all'essenziale: due costruttori che definiscono il modo in cui viene creato l'oggetto **MyException**. Nel secondo costruttore, il costruttore della classe di base con un argomento **String** viene esplicitamente chiamato con la parola chiave **super**.

Nei gestori viene chiamato **printStackTrace()**, uno dei metodi di **Throwable** da cui eredita **Exception**. Come potete vedere dall'output, questo metodo fornisce informazioni sulla sequenza di metodi che sono stati chiamati fino al punto in cui si è verificata l'eccezione. In questo caso specifico le informazioni sono trasmesse a **System.out** e automaticamente intercettate e visualizzate. Tuttavia, se chiamate la versione predefinita

```
e.printStackTrace();
```

le informazioni saranno indirizzate al flusso di standard error.

**Esercizio 1** (2) Create una classe con un metodo **main()** che solleva un oggetto di classe **Exception** all'interno di un blocco **try**, e assegnate un argomento di tipo **String** al costruttore di **Exception**. Intercettate l'eccezione all'interno di un'istruzione **catch** e visualizzate l'argomento **String**; infine, aggiungete un'istruzione **finally** e visualizzate un messaggio che dimostri che siete esattamente in quel punto (troverete la spiegazione di questa istruzione più avanti nel capitolo).

**Esercizio 2** (1) Definite un riferimento di oggetto e inizializzatelo a **null**, provando a chiamare un metodo tramite questo riferimento. Poi includete il codice in un blocco **try-catch** per intercettare l'eccezione.

**Esercizio 3** (1) Scrivete il codice per generare e intercettare un'eccezione di tipo **ArrayIndexOutOfBoundsException**.

**Esercizio 4** (2) Create una classe di eccezioni personalizzata utilizzando la parola chiave **extends**; per questa classe scrivete un costruttore che accetti un argomento di tipo **String** e lo registri all'interno dell'oggetto con un riferimento a un oggetto **String**, quindi realizzate un metodo che visualizza la stringa registrata. Infine create un blocco **try-catch** per provare il funzionamento della vostra nuova eccezione.

**Esercizio 5** (3) Generate un vostro comportamento di tipo "ripristino" utilizzando un ciclo **while** che si ripete fino a quando non sia sollevata alcuna eccezione.

## *Eccezioni e log*

Ricorrendo alla funzione **java.util.logging** avete anche la possibilità di tenere un log dell'output.

I dettagli completi della funzionalità di logging rientrano tra gli argomenti del supplemento disponibile all'indirizzo *http://MindView.net/Books/BetterJava*; in ogni caso il meccanismo di log di base è abbastanza immediato da poter essere applicato in questo contesto.

```
//: exceptions/LoggingExceptions.java
// Un'eccezione registrata tramite un Logger.
import java.util.logging.*;
import java.io.*;

class LoggingException extends Exception {
  private static Logger logger =
    Logger.getLogger("LoggingException");
```

```
  public LoggingException() {
    StringWriter trace = new StringWriter();
    printStackTrace(new PrintWriter(trace));
    logger.severe(trace.toString());
  }
}

public class LoggingExceptions {
  public static void main(String[] args) {
    try {
      throw new LoggingException();
    } catch(LoggingException e) {
      System.err.println("Caught " + e);
    }
    try {
      throw new LoggingException();
    } catch(LoggingException e) {
      System.err.println("Caught " + e);
    }
  }
} /* Output: (85% match)
Aug 30, 2005 4:02:31 PM LoggingException <init>
SEVERE: LoggingException
        at LoggingExceptions.main(LoggingExceptions.java:19)
Caught LoggingException
Aug 30, 2005 4:02:31 PM LoggingException <init>
SEVERE: LoggingException
        at LoggingExceptions.main(LoggingExceptions.java:24)

Caught LoggingException
*///:~
```

Il metodo **static Logger.getLogger()** genera un oggetto **Logger** associato all'argomento **String**, generalmente costituito dal nome del pacchetto e dalla classe cui si riferiscono gli errori: questo oggetto invia l'output a **System.err**. Il modo più facile per registrare su un **Logger** è chiamare il metodo associato al livello del messaggio di log, **severe()** in questo caso specifico. Per produrre

la **String** per il messaggio di log occorre risalire al punto in cui si è verificata l'eccezione, tuttavia **printStackTrace()** non produce una stringa in modo predefinito; dovrete quindi utilizzare il metodo **printStackTrace()** sovraccarico, che accetta come argomento un oggetto **java.io.PrintWriter**: questo meccanismo verrà illustrato in dettaglio nel Capitolo 6. Se passate al costruttore di **PrintWriter** un oggetto **java.io.StringWriter** l'output potrà essere estratto come **String** chiamando il metodo **toString()**.

Per quanto la tecnica adottata da **LoggingException** sia pratica, poiché costruisce l'intera infrastruttura di log all'interno dell'eccezione stessa, e può così funzionare automaticamente senza l'intervento del programmatore client, è più comune trovarsi nella necessità di intercettare e registrare le eccezioni di altri programmatori, al fine di generare il messaggio di log nel gestore di eccezioni.

```
//: exceptions/LoggingExceptions2.java
// Registrazione in log delle eccezioni intercettate.
import java.util.logging.*;
import java.io.*;

public class LoggingExceptions2 {
  private static Logger logger =
    Logger.getLogger("LoggingExceptions2");
  static void logException(Exception e) {
    StringWriter trace = new StringWriter();
    e.printStackTrace(new PrintWriter(trace));
    logger.severe(trace.toString());
  }
  public static void main(String[] args) {
    try {
      throw new NullPointerException();
    } catch(NullPointerException e) {
      logException(e);
    }
  }
} /* Output: (90% match)
Aug 30, 2005 4:07:54 PM LoggingExceptions2 logException
SEVERE: java.lang.NullPointerException
        at LoggingExceptions2.main(LoggingExceptions2.java:16)
*///:~
```

Il processo di creazione delle vostre eccezioni personalizzate può essere ulteriormente potenziato, aggiungendo altri costruttori e membri.

```
//: exceptions/ExtraFeatures.java
// Miglioramenti alle classi di eccezioni.
import static net.mindview.util.Print.*;

class MyException2 extends Exception {
  private int x;
  public MyException2() {}
  public MyException2(String msg) { super(msg); }
  public MyException2(String msg, int x) {
    super(msg);
    this.x = x;
  }
  public int val() { return x; }
  public String getMessage() {
    return "Detail Message: "+ x + " "+ super.getMessage();
}
}

public class ExtraFeatures {
  public static void f() throws MyException2 {
    print("Throwing MyException2 from f()");
    throw new MyException2();
  }
  public static void g() throws MyException2 {
    print("Throwing MyException2 from g()");
    throw new MyException2("Originated in g()");
  }
  public static void h() throws MyException2 {
    print("Throwing MyException2 from h()");
    throw new MyException2("Originated in h()", 47);
  }
  public static void main(String[] args) {
    try {
      f();
```

```
      } catch(MyException2 e) {
        e.printStackTrace(System.out);
      }
      try {
        g();
      } catch(MyException2 e) {
        e.printStackTrace(System.out);
}
      try {
        h();
      } catch(MyException2 e) {
        e.printStackTrace(System.out);
        System.out.println("e.val() = " + e.val());
      }
    }
  } /* Output:
  Throwing MyException2 from f()
  MyException2: Detail Message: 0 null
          at ExtraFeatures.f(ExtraFeatures.java:22)
          at ExtraFeatures.main(ExtraFeatures.java:34)
  Throwing MyException2 from g()
  MyException2: Detail Message: 0 Originated in g()
          at ExtraFeatures.g(ExtraFeatures.java:26)
          at ExtraFeatures.main(ExtraFeatures.java:39)
  Throwing MyException2 from h()
  MyException2: Detail Message: 47 Originated in h()
          at ExtraFeatures.h(ExtraFeatures.java:30)
          at ExtraFeatures.main(ExtraFeatures.java:44)
  e.val() = 47
  *///:~
```

Nel codice è stato aggiunto un campo **x**, con un metodo che legge questo valore e un costruttore supplementare che lo imposta. Inoltre, **Throwable.getMessage()** è stato sovrascritto per produrre un messaggio con dettagli più interessanti. Il metodo **getMessage()** è analogo a **toString()** per le classi di eccezioni.

Poiché un'eccezione non è altro che un tipo di oggetto, nulla vi vieta di continuare in questo processo di perfezionamento delle classi di eccezioni. Tenete presente, tuttavia, che tutta la vostra attività di miglioramento potrebbe risultare vana quando i programmatori client utilizzeranno i vostri package:

molto spesso essi si limitano a cercare l'eccezione da sollevare, come se avessero a che fare con una qualsiasi libreria di eccezioni Java.

**Esercizio 6** (1) Create due classi di eccezioni, ciascuna delle quali esegue automaticamente il logging, e dimostratene il funzionamento.

**Esercizio 7** (1) Modificate l'Esercizio 3 in modo che l'istruzione **catch** registri in log i risultati.

## Specifiche di eccezione

In Java è sempre opportuno che informiate il programmatore client, che utilizzerà il metodo da voi creato, delle eccezioni che potrebbero essere prodotte dal metodo stesso: questo suggerimento è molto pratico poiché in questo modo il chiamante potrà conoscere esattamente il tipo di codice che dovrà scrivere per intercettare tutte le potenziali eccezioni. Naturalmente non vi sarebbe alcun problema se il codice sorgente fosse sempre disponibile, poiché in tal caso il programmatore che si serve della libreria di classi creata da voi non dovrebbe fare altro che cercare le dichiarazioni **throw**: tenete presente, tuttavia, che non sempre con una libreria vengono messi a disposizione i file sorgente. Per prevenire ogni possibile problema Java fornisce una sintassi che vi costringe a rispettare, affinché chi utilizza una certa classe possa capire quali eccezioni vengono sollevate da un suo metodo e successivamente gestirle: si tratta delle cosiddette *specifiche di eccezione* (*exception specification*), che nelle dichiarazioni dei metodi figurano subito dopo l'elenco degli argomenti.

Le specifiche di eccezione richiedono l'utilizzo della parola chiave **throws**, seguita dall'elenco di tutti i possibili tipi di eccezione. Pertanto la definizione del vostro metodo potrebbe essere simile alla seguente:

```
void f() throws TooBig, TooSmall, DivZero { //...
```

Tuttavia, se scrivete:

```
void f() { // ...
```

indicherete quindi che il metodo non produrrà eccezioni, tranne quelle ereditate da **RuntimeException**, che, come vedrete in seguito, possono essere sollevate in qualunque occasione senza specifiche di eccezione.

L'utilizzo delle specifiche di eccezione consente, in un certo senso, di "mentire". Se il codice del vostro metodo genera eccezioni che il metodo stesso non gestisce, il compilatore rileva questo conflitto segnalandovi la necessità di

gestire l'eccezione oppure di indicare mediante una specifica che il vostro metodo può produrre una determinata eccezione. Costringendovi a implementare in modo rigoroso le specifiche di eccezione, Java garantisce che in fase di compilazione sia mantenuto un livello minimo di correttezza nelle eccezioni.

Il modo in cui potete "mentire" è sostenendo di sollevare un'eccezione, mentre in realtà questo non avviene: il compilatore prenderà per buona la vostra affermazione, obbligando gli utenti del vostro metodo a considerarlo come se effettivamente producesse quell'eccezione. Tale comportamento ha per effetto quello di creare una sorta di "segnalibro" per quell'eccezione, che vi consentirà di sollevarla in un momento successivo senza dover intervenire modificando il codice esistente. La stessa tecnica è importante anche nella creazione di classi di base **abstract** e di interfacce, le cui classi o implementazioni derivate potrebbero avere necessità di produrre eccezioni.

Le eccezioni controllate e imposte al momento della compilazione sono denominate *eccezioni controllate* (*checked exception*).

> **Esercizio 8** (1) Scrivete una classe con un metodo che solleva un'eccezione del tipo generato nell'Esercizio 4; provate a compilarla senza specifiche di eccezione e prendete nota di quanto vi segnalerà il compilatore. Aggiungete la specifica di eccezione opportuna e testate la vostra classe e la relativa eccezione all'interno del blocco **try-catch**.

## Intercettare qualsiasi eccezione

Avete la possibilità di generare un gestore che intercetti qualunque tipo di eccezione. Per fare questo intercettate il tipo di eccezione della classe di base, **Exception**. Tenete presente che esistono altri tipi di eccezioni di base; in ogni caso **Exception** è considerata praticamente il punto di partenza di qualsiasi attività di programmazione:

```
catch(Exception e) {
  System.out.println("Caught an exception");
}
```

Questa tecnica intercetterà qualsiasi eccezione: di conseguenza, se ritenete di servirvene, dovrete inserire il codice dopo l'elenco dei gestori, per evitare che possa attivarsi prima di qualsiasi altro gestore di eccezioni che segua.

Essendo il capostipite di tutte le classi di eccezione che sono importanti per il programmatore, **Exception** non fornisce molte informazioni specifiche sul-

l'eccezione, che tuttavia possono essere ottenute chiamando i metodi che derivano dal suo tipo di base **Throwable**.

**String getMessage()**

**String getLocalizedMessage()**

Fornisce il messaggio dettagliato di descrizione dell'errore (**getMessage()**) o un messaggio adattato alle specifiche impostazioni linguistiche del sistema (**getLocalizedMessage()**).

**String toString()**

Restituisce una descrizione breve del tipo **Throwable**, che include l'eventuale messaggio dettagliato.

**void printStackTrace()**

**void printStackTrace(PrintStream)**

**void printStackTrace(java.io.PrintWriter)**

Visualizza il tipo di eccezione **Throwable** e il tracciato dello stack (*stack trace*) della chiamata a **Throwable**. Lo stack di chiamata mostra la sequenza delle chiamate di metodo che hanno portato al verificarsi dell'eccezione. La prima versione invia il risultato al dispositivo di standard error, mentre la seconda e la terza inviano il risultato a un flusso di vostra scelta: nel Capitolo 6 vedrete i motivi che giustificano l'esistenza di due tipi di flusso.

**Throwable fillInStackTrace()**

Nell'ambito dell'oggetto **Throwable**, registra le informazioni sullo stato corrente dei livelli dello stack. Come vedrete tra breve, questo metodo è utile quando un'applicazione solleva ripetutamente un errore o un'eccezione.

Oltre a quelli elencati, sono disponibili altri metodi forniti da **Object**, il tipo di base di **Throwable** e di qualsiasi altro oggetto. Troverete pratico per le eccezioni **getClass()**, che restituisce un oggetto che rappresenta la classe dell'oggetto corrente: potrete poi richiedere il nome di questo oggetto **Class** ricorrendo al metodo **getName()**, ottenendo anche le informazioni sul pacchetto, o al metodo **getSimpleName()**, che restituisce soltanto il nome della classe.

L'esempio seguente mostra come utilizzare i metodi di base di **Exception**.

```
//: exceptions/ExceptionMethods.java
// Dimostrazione dei metodi di Exception.
import static net.mindview.util.Print.*;

public class ExceptionMethods {
  public static void main(String[] args) {
    try {
      throw new Exception("My Exception");
    } catch(Exception e) {
      print("Caught Exception");
      print("getMessage(): " + e.getMessage());
      print("getLocalizedMessage(): " +
        e.getLocalizedMessage());
      print("toString(): " + e);
      print("printStackTrace(): ");
      e.printStackTrace(System.out);
    }
  }
} /* Output:
Caught Exception
getMessage(): My Exception
getLocalizedMessage(): My Exception
toString(): java.lang.Exception: My Exception
printStackTrace():
java.lang.Exception: My Exception
        at ExceptionMethods.main(ExceptionMethods.java:8)
*///:~
```

Potete osservare che i metodi forniscono progressivamente più informazioni, in quanto ognuno è il sovrainsieme del metodo precedente.

**Esercizio 9** (2) Generate tre nuovi tipi di eccezione e scrivete una classe con un metodo che le sollevi tutte e tre. In **main()** chiamate il metodo, ma utilizzando una sola istruzione **catch** che intercetti tutti i tre tipi di eccezione.

## *Tracciamento dello stack*

Le informazioni fornite da **printStackTrace()** sono accessibili anche direttamente tramite il metodo **getStackTrace()**, che restituisce un array degli elementi del tracciato dello stack, ciascuno dei quali rappresenta una chiamata. L'elemento zero corrisponde alla parte superiore dello stack ed è l'ultima chiamata di metodo presente nella sequenza, vale a dire il punto in cui è stato creato e sollevato **Throwable**; l'ultimo elemento dell'array, nella parte inferiore dello stack, è invece la prima chiamata di metodo nella sequenza. Questo programma fornisce una semplice dimostrazione.

```
//: exceptions/WhoCalled.java
// Accesso programmatico alle informazioni dello stack trace.

public class WhoCalled {
  static void f() {
    // Genera un'eccezione per popolare lo stack trace
    try {
      throw new Exception();
    } catch (Exception e) {
      for(StackTraceElement ste : e.getStackTrace())
        System.out.println(ste.getMethodName());
    }
  }
  static void g() { f(); }
  static void h() { g(); }
  public static void main(String[] args) {
    f();
    System.out.println("--------------------------------");
    g();
    System.out.println("--------------------------------");
    h();
  }
} /* Output:
f
main
--------------------------------
f
```

```
g
main
--------------------------------
f
g
h
main
*///:~
```

Il codice si limita a visualizzare il nome di metodo ma è anche possibile visualizzare l'intero **StackTraceElement**, che contiene informazioni supplementari.

## *Eccezioni sollevate ripetutamente*

A volte vorrete produrre di nuovo un'eccezione appena intercettata, in particolare quando utilizzate **Exception** per intercettare qualsiasi tipo di eccezione. Dal momento che avete già il riferimento all'eccezione corrente, non dovete fare altro che produrre nuovamente tale riferimento:

```
catch(Exception e) {
  System.out.println("An exception was thrown");
  throw e;
}
```

Sollevare più volte un'eccezione fa sì che essa venga indirizzata al gestore di eccezioni nel successivo contesto di livello più elevato. Tutte le altre istruzioni **catch** dello stesso blocco **try** vengono ignorate; inoltre l'intero oggetto di eccezione è mantenuto, in modo che il gestore di livello più elevato che intercetta lo specifico tipo di eccezione possa estrarne tutte le informazioni necessarie.

Se vi limitate a produrre nuovamente l'eccezione corrente, le informazioni che otterrete su di essa con **printStackTrace()** si riferiranno al punto originario dell'eccezione, non al punto in cui essa viene prodotta di nuovo. Se desiderate inserire nuove informazioni sul tracciato dello stack potete chiamare **fillInStackTrace()**, che restituisce un oggetto **Throwable** ottenuto integrando le informazioni dello stack corrente nel vecchio oggetto di eccezione, come indicato di seguito.

```
//: exceptions/Rethrowing.java
// Dimostrazione di fillInStackTrace()
```

```
public class Rethrowing {
  public static void f() throws Exception {
    System.out.println("originating the exception in f()");
    throw new Exception("thrown from f()");
}
  public static void g() throws Exception {
    try {
      f();
    } catch(Exception e) {
      System.out.println("Inside g(), e.printStackTrace()");
      e.printStackTrace(System.out);
      throw e;
    }
  }
  public static void h() throws Exception {
    try {
      f();
    } catch(Exception e) {
      System.out.println("Inside h(), e.printStackTrace()");
      e.printStackTrace(System.out);
      throw (Exception)e.fillInStackTrace();
    }
  }
  public static void main(String[] args) {
    try {
      g();
    } catch(Exception e) {
      System.out.println("main: printStackTrace()");
      e.printStackTrace(System.out);
    }
    try {
      h();
    } catch(Exception e) {
      System.out.println("main: printStackTrace()");
      e.printStackTrace(System.out);
    }
```

```
  }
} /* Output:
originating the exception in f()
Inside g(),e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:7)
        at Rethrowing.g(Rethrowing.java:11)
        at Rethrowing.main(Rethrowing.java:29)
main: printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:7)
        at Rethrowing.g(Rethrowing.java:11)
at Rethrowing.main(Rethrowing.java:29)
originating the exception in f()
Inside h(),e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:7)
        at Rethrowing.h(Rethrowing.java:20)
        at Rethrowing.main(Rethrowing.java:35)
main: printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.h(Rethrowing.java:24)
        at Rethrowing.main(Rethrowing.java:35)
*///:~
```

La riga in cui viene chiamato **fillInStackTrace()** diventa il nuovo punto di origine dell'eccezione.

È anche possibile sollevare un'eccezione diversa da quella intercettata, ottenendo in tal caso un effetto analogo all'utilizzo di **fillInStackTrace()**, nel quale le informazioni sul punto originale in cui è avvenuta l'eccezione vengono perse e tutto ciò che rimane sono le informazioni relative al nuovo **throw**.

```
//: exceptions/RethrowNew.java
// Solleva di nuovo un oggetto diverso da quello intercettato
// in precedenza.

class OneException extends Exception {
  public OneException(String s) { super(s); }
```

```
}

class TwoException extends Exception {
  public TwoException(String s) { super(s); }
}

public class RethrowNew {
  public static void f() throws OneException {
    System.out.println("originating the exception in f()");
    throw new OneException("thrown from f()");
  }
  public static void main(String[] args) {
    try {
      try {
        f();
      } catch(OneException e) {
        System.out.println(
          "Caught in inner try, e.printStackTrace()");
        e.printStackTrace(System.out);
throw new TwoException("from inner try");
      }
    } catch(TwoException e) {
      System.out.println(
        "Caught in outer try, e.printStackTrace()");
      e.printStackTrace(System.out);
    }
  }
} /* Output:
originating the exception in f()
Caught in inner try, e.printStackTrace()
OneException: thrown from f()
        at RethrowNew.f(RethrowNew.java:15)
        at RethrowNew.main(RethrowNew.java:20)
Caught in outer try, e.printStackTrace()
TwoException: from inner try
        at RethrowNew.main(RethrowNew.java:25)
*///:~
```

L'eccezione finale sa soltanto che proviene dal blocco **try** interno e non da **f()**.

Non dovete mai preoccuparvi di eliminare l'eccezione precedente né qualsiasi altra: sono tutti oggetti basati sull'heap e creati con **new**, che il garbage collector avrà cura di eliminare automaticamente.

## *Concatenamento di eccezioni*

Spesso si vuole intercettare un'eccezione e sollevarne un'altra, pur mantenendo le informazioni sull'eccezione originale: la tecnica che consente questa operazione è chiamata *concatenamento di eccezioni*. Prima di JDK 1.4, per conservare le informazioni originali dell'eccezione occorreva scrivere codice specifico, ma ora i costruttori di tutte le sottoclassi di **Throwable** accettano un oggetto *cause*: questa "causa" va intesa nel significato di "eccezione originante" e permette di mantenere lo stack trace nelle condizioni originarie, sebbene generi e sollevi una nuova eccezione.

È interessante notare che le sole sottoclassi di **Throwable** che offrono l'argomento **Cause** nel loro costruttore sono le tre classi di eccezione fondamentali: **Error** (utilizzata dalla JVM per segnalare gli errori di sistema), **Exception** e **RuntimeException**. Se volete concatenare qualunque altro tipo di eccezione dovrete ricorrere al metodo **initCause()** anziché al costruttore.

Di seguito è mostrato un esempio che permette di aggiungere dinamicamente i campi a un oggetto **DynamicFields** in fase di esecuzione.

```
//: exceptions/DynamicFields.java
// Una classe che aggiunge dinamicamente campi a se stessa.
// Dimostrazione del concatenamento di eccezioni.
import static net.mindview.util.Print.*;

class DynamicFieldsException extends Exception {}

public class DynamicFields {
  private Object[][] fields;
  public DynamicFields(int initialSize) {
    fields = new Object[initialSize][2];
    for(int i = 0; i < initialSize; i++)
      fields[i] = new Object[] { null, null };
  }
  public String toString() {
```

```
    StringBuilder result = new StringBuilder();
    for(Object[] obj : fields) {
      result.append(obj[0]);
      result.append(": ");
      result.append(obj[1]);
      result.append("\n");
    }
    return result.toString();
  }
  private int hasField(String id) {
    for(int i = 0; i < fields.length; i++)
      if(id.equals(fields[i][0]))
        return i;
    return -1;
  }
  private int
  getFieldNumber(String id) throws NoSuchFieldException {
    int fieldNum = hasField(id);
    if(fieldNum == -1)
      throw new NoSuchFieldException();
    return fieldNum;
  }
  private int makeField(String id) {
    for(int i = 0; i < fields.length; i++)
      if(fields[i][0] == null) {
        fields[i][0] = id;
return i;
      }
    // Nessun campo vuoto. Ne viene aggiunto uno:
    Object[][] tmp = new Object[fields.length + 1][2];
    for(int i = 0; i < fields.length; i++)
      tmp[i] = fields[i];
    for(int i = fields.length; i < tmp.length; i++)
      tmp[i] = new Object[] { null, null };
    fields = tmp;
    // Chiamata ricorsiva con campi espansi:
```

```
      return makeField(id);
    }
    public Object
    getField(String id) throws NoSuchFieldException {
      return fields[getFieldNumber(id)][1];
    }
    public Object setField(String id, Object value)
    throws DynamicFieldsException {
      if(value == null) {
        // La maggior parte delle eccezioni non ha un
        // costruttore con "cause".
        // In questi casi occorre utilizzare initCause(),
        // disponibile in tutte le sottoclassi Throwable.
        DynamicFieldsException dfe =
          new DynamicFieldsException();
        dfe.initCause(new NullPointerException());
        throw dfe;
      }
      int fieldNumber = hasField(id);
      if(fieldNumber == -1)
        fieldNumber = makeField(id);
      Object result = null;
      try {
        result = getField(id); // Ottiene il vecchio valore
      } catch(NoSuchFieldException e) {
        // Utilizza un costruttore che accetta "cause":
        throw new RuntimeException(e);
      }
      fields[fieldNumber][1] = value;
      return result;
    }
    public static void main(String[] args) {
      DynamicFields df = new DynamicFields(3);
      print(df);
      try {
  df.setField("d", "A value for d");
```

```
      df.setField("number", 47);
      df.setField("number2", 48);
      print(df);
      df.setField("d", "A new value for d");
      df.setField("number3", 11);
      print("df: " + df);
      print("df.getField(\"d\") : " + df.getField("d"));
      Object field = df.setField("d", null); // Eccezione
    } catch(NoSuchFieldException e) {
      e.printStackTrace(System.out);
    } catch(DynamicFieldsException e) {
      e.printStackTrace(System.out);
    }
  }
} /* Output:
null: null
null: null
null: null

d: A value for d
number: 47
number2: 48

df: d: A new value for d
number: 47
number2: 48
number3: 11

df.getField("d") : A new value for d
DynamicFieldsException
        at DynamicFields.setField(DynamicFields.java:64)
        at DynamicFields.main(DynamicFields.java:94)
Caused by: java.lang.NullPointerException
        at DynamicFields.setField(DynamicFields.java:66)
        ... 1 more
*///:~
```

Ogni oggetto **DynamicFields** contiene un array di coppie **Object-Object**. Il primo oggetto è l'identificativo di campo (una **String**) e il secondo è il valore del campo, che può essere qualunque tipo tranne un primitivo non incluso in un wrapper. Quando generate l'oggetto dovete prevedere il numero di campi che occorreranno. Il metodo **setField()** trova il campo esistente con il nome passato come identificativo oppure ne genera uno nuovo, e in entrambi i casi vi inserisce il valore. In caso di esaurimento dello spazio disponibile, **setField()** genera un nuovo array, più grande di una posizione rispetto al precedente, e vi copia gli elementi dell'array precedente. Se provate a inserire un valore **null**, **setField()** solleverà una **DynamicFieldsException** creandone una nuova istanza (**new DynamicFieldsException()**) e utilizzando **initCause()** per inserire una **NullPointerException** come *cause*.

Come valore di ritorno **setField()** intercetta anche il vecchio valore di **d** corrispondente alla posizione del campo, utilizzando **getField()**, che potrebbe produrre un'eccezione **NoSuchFieldException**. Se il programmatore client chiama **getField()** sarà responsabile della gestione di **NoSuchFieldException**, ma se questa eccezione viene sollevata all'interno di **setField()** si tratterà di un errore di programmazione, pertanto **NoSuchFieldException** verrà convertita in **RuntimeException** utilizzando il costruttore che accetta un argomento *cause*.

Notate che **toString()** impiega un oggetto **StringBuilder** per creare i suoi risultati. Nel Volume 1, Capitolo 12 è stato introdotto questo oggetto, impiegato generalmente per scrivere un metodo **toString()** che richiede un'iterazione, come nell'esempio in questione.

> **Esercizio 10** (2) Generate una classe con due metodi, **f()** e **g()**. In **g()** sollevate un'eccezione di un nuovo tipo, che dovrete definire; in **f()** chiamate **g()**, intercettatene l'eccezione e nell'istruzione **catch** producete un'altra eccezione, di un tipo diverso che definirete. In **main()** verificate il funzionamento del vostro codice.
>
> **Esercizio 11** (1) Ripetete l'esercizio precedente, ma all'interno dell'istruzione **catch** inglobate l'eccezione di **g()** in una **RuntimeException**.

## Eccezioni Java standard

La classe **Throwable** di Java descrive tutto ciò che può essere sollevato come eccezione. Esistono due tipi generali di oggetti **Throwable**, dove per "tipi di oggetti" si intende "derivati da".

**Error** rappresenta gli errori di compilazione e di sistema che di norma non vengono intercettati, tranne che in occasioni molto speciali. **Exception** è il tipo

di base che può essere sollevato da un qualsiasi metodo di classe delle librerie Java e dai vostri metodi ed eventi che si verificassero in fase di esecuzione. Pertanto, il tipo che normalmente interessa il programmatore Java è **Exception**.

Il modo migliore per ottenere una panoramica delle eccezioni è consultare la documentazione JDK. È opportuno che lo facciate non appena inizierete a distinguere le varie eccezioni, anche se presto vi renderete conto che non vi è alcuna differenza particolare tra un'eccezione e l'altra, a parte il nome.

Tenete presente, inoltre, che il numero di eccezioni in Java continua a espandersi, ed è quindi inutile elencarle in un libro. Anche ogni nuova libreria che otterrete da un produttore esterno avrà probabilmente eccezioni personalizzate. In ogni caso, il concetto fondamentale è comprendere il meccanismo delle eccezioni e come gestirle.

L'idea di base è che il nome dell'eccezione rappresenti il problema che si è verificato e che tale nome sia relativamente autoesplicativo. Non tutte le eccezioni sono definite in **java.lang**; alcune vengono generate per supportare altre librerie, quali **util**, **net** e **io**, come potrete vedere dai loro nomi di classe completi o dall'elemento da cui ereditano: per esempio, tutte le eccezioni di input/output (I/O) sono ereditate da **java.io.IOException**.

## Caso speciale: RuntimeException

Il primo esempio in questo capitolo è il seguente:

```
if(t == null)
  throw new NullPointerException();
```

Potrebbe sconvolgervi l'idea di dover controllare la presenza di **null** in ogni riferimento che passerete a un metodo, dal momento che non potete essere certi che il chiamante abbia passato un riferimento valido. Fortunatamente questo non è necessario poiché il compito rientra tra i controlli standard svolti da Java in fase di esecuzione: se una chiamata fa riferimento a **null** Java solleverà automaticamente una **NullPointerException**. Pertanto il frammento di codice precedente è sempre superfluo, benché nulla vi vieti di effettuare altri controlli per cautelarvi dal verificarsi di una **NullPointerException**.

Questa categoria comprende un intero gruppo di tipi di eccezione, tutte sempre prodotte automaticamente da Java e che non avrete bisogno di includere nelle vostre specifiche di eccezione. Per praticità tali eccezioni sono raggruppate in un'unica classe di base chiamata **RuntimeException**, che è un perfetto esempio di ereditarietà poiché imposta una famiglia di tipi che condividono comportamenti e caratteristiche. Inoltre, non è mai necessario scrivere

una specifica di eccezione per definire che un metodo potrebbe sollevare una **RuntimeException** (o qualsiasi tipo ereditato da **RuntimeException**), poiché si tratta di eccezioni non controllate (*unchecked exception*). Poiché una **RuntimeException** indica un errore nel codice, di solito non la intercetterete: verrà gestita automaticamente. Se foste costretti a controllare la presenza di **RuntimeException**, il vostro codice potrebbe diventare troppo complesso. Tenete presente che sebbene normalmente non intercetterete le **RuntimeException**, nei vostri pacchetti potreste comunque decidere di sollevarne qualcuna.

Che cosa accade quando non si intercettano tali eccezioni? Dal momento che il compilatore non forza specifiche di eccezione per esse, è abbastanza verosimile che una **RuntimeException** possa filtrare dal vostro metodo **main()** senza essere intercettata. Per analizzare che cosa accade in questi casi provate a eseguire l'esempio seguente.

```
//: exceptions/NeverCaught.java
// Come ignorare le RuntimeException.
// {ThrowsException}

public class NeverCaught {
  static void f() {
    throw new RuntimeException("From f()");
  }
  static void g() {
    f();
  }
  public static void main(String[] args) {
    g();
  }
} ///:~
```

Potete vedere che una **RuntimeException** (o qualsiasi eccezione ereditata da essa) è un caso speciale, poiché il compilatore non richiede una specifica di eccezione per questi tipi. L'output è inviato a **System.err**:

```
Exception in thread "main" java.lang.RuntimeException: From f()
        at NeverCaught.f(NeverCaught.java:7)
        at NeverCaught.g(NeverCaught.java:10)
        at NeverCaught.main(NeverCaught.java:13)
```

Quindi la risposta è: se una **RuntimeException** esce da **main()** senza essere intercettata, all'uscita del programma viene chiamato il metodo **printStack-Trace()** per quell'eccezione.

Tenete presente che nel vostro codice potrete ignorare soltanto le eccezioni di tipo **RuntimeException** (e sottoclassi), dal momento che il compilatore si assicura che tutte le eccezioni controllate siano gestite. Il concetto è che una **RuntimeException** rappresenta un errore di programmazione, ossia uno di quelli elencati di seguito.

1. Un errore che non potete prevedere, per esempio un riferimento **null** che è fuori dal vostro controllo.
2. Un errore che come programmatori avreste dovuto controllare nel codice, per esempio una **ArrayIndexOutOfBoundsException** a fronte della quale avreste dovuto verificare le dimensioni dell'array. Un'eccezione di tipo 1 spesso si trasforma in un problema di tipo 2.

In questo caso potete vedere quale beneficio rappresenti il fatto di disporre delle eccezioni, che sono di aiuto nel processo di debugging.

È interessante rilevare che non è possibile classificare la gestione delle eccezioni in Java come uno strumento specifico per un determinato scopo: certo, è destinata a gestire quegli errori fastidiosi che possono verificarsi in fase di esecuzione a causa di eventi fuori dal controllo del vostro codice, tuttavia è anche essenziale per alcuni tipi di errori di programmazione che il compilatore non è in grado di rilevare.

> **Esercizio 12** (3) Modificate **innerclasses/Sequence.java** in modo che sollevi un'eccezione opportuna quando provate a inserirvi troppi elementi.

## Esecuzione delle operazioni di cleanup con finally

Spesso avrete l'esigenza di eseguire una determinata porzione di codice a prescindere se nel blocco **try** viene prodotta un'eccezione oppure no.

Comunemente si tratterà di operazioni diverse dal recupero della memoria, che è presa in carico dal garbage collector. Per ottenere questo effetto, utilizzate un'istruzione **finally** al termine di tutti i gestori di eccezioni.[3]

3. La gestione delle eccezioni in C++ non dispone dell'istruzione **finally** poiché questo linguaggio affida ai distruttori il compito di eseguire questo tipo di cleanup.

Lo schema completo di una sezione di gestione delle eccezioni è il seguente.

```
try {
  // La cosiddetta guarded region, per le attivita' pericolose
  // che potrebbero sollevare A, B o C
} catch(A a1) {
  // Gestore per la situazione A
} catch(B b1) {
  // Gestore per la situazione B
} catch(C c1) {
  // Gestore per la situazione C
} finally {
  // Attivita' che avvengono in qualsiasi occasione
}
```

Per dimostrare che l'istruzione **finally** funziona in qualsiasi caso eseguite il programma seguente.

```
//: exceptions/FinallyWorks.java
// L'istruzione finally viene sempre eseguita.

class ThreeException extends Exception {}

public class FinallyWorks {
  static int count = 0;
  public static void main(String[] args) {
    while(true) {
      try {
        // La prima volta il post-incremento e' zero:
        if(count++ == 0)
          throw new ThreeException();
        System.out.println("No exception");
      } catch(ThreeException e) {
        System.out.println("ThreeException");
      } finally {
        System.out.println("In finally clause");
        if(count == 2) break; // fuori da "while"
      }
    }
```

```
  }
} /* Output:
ThreeException
In finally clause
No exception
In finally clause
*///:~
```

Dall'output è evidente che l'istruzione **finally** viene sempre eseguita.

Questo programma fornisce inoltre un suggerimento per fare fronte al fatto che le eccezioni Java, come si è detto, non permettono di ritornare al punto in cui è stata sollevata l'eccezione. Se disponete il vostro blocco **try** all'interno di un'iterazione potrete impostare una condizione che dovrà essere soddisfatta prima di continuare il programma. Potete anche aggiungere un contatore **static** o qualche altro dispositivo per consentire al ciclo di provare vari approcci prima di rinunciare a proseguire il programma: in questo modo i vostri programmi risulteranno decisamente più robusti.

## *A che cosa serve finally?*

In un linguaggio che non offre funzionalità particolari di garbage collection né chiamate automatiche al distruttore, **finally** è importante poiché permette al programmatore di garantire il rilascio della memoria indipendentemente da ciò che accade nel blocco **try**.[4] Java tuttavia può contare sulla garbage collection, pertanto il rilascio della memoria non è quasi mai un problema, e inoltre non ha distruttori da chiamare. Quindi, quando utilizzare **finally** in Java?

L'istruzione **finally** è necessaria quando occorra riportare alle condizioni originali un componente diverso dalla memoria. Potrebbe trattarsi di un file o un collegamento di rete rimasto aperto, di qualcosa tracciato sullo schermo o persino di un interruttore esterno, come mostra il seguente esempio.

```
//: exceptions/Switch.java
import static net.mindview.util.Print.*;
public class Switch {
  private boolean state = false;
```

4. Un distruttore è una funzione che viene sempre chiamata quando un oggetto non è più utilizzato. Si sa sempre esattamente dove e quando viene chiamato il distruttore. C++ offre chiamate automatiche al distruttore, mentre C#, più simile a Java, dispone di una funzionalità di distruzione automatica.

```
  public boolean read() { return state; }
  public void on() { state = true; print(this); }
  public void off() { state = false; print(this); }
  public String toString() { return state ? "on" : "off"; }
} ///:~

//: exceptions/OnOffException1.java
public class OnOffException1 extends Exception {} ///:~

//: exceptions/OnOffException2.java
public class OnOffException2 extends Exception {} ///:~

//: exceptions/OnOffSwitch.java
// Perche' usare finally?

public class OnOffSwitch {
  private static Switch sw = new Switch();
public static void f()
  throws OnOffException1,OnOffException2 {}
  public static void main(String[] args) {
    try {
      sw.on();
      // Codice che puo' sollevare eccezioni...
      f();
      sw.off();
    } catch(OnOffException1 e) {
      System.out.println("OnOffException1");
      sw.off();
    } catch(OnOffException2 e) {
      System.out.println("OnOffException2");
      sw.off();
    }
  }
} /* Output:
on
off
*///:~
```

Lo scopo di questo codice è garantire che l'interruttore sia disattivato al completamento di **main()**, di conseguenza il metodo **sw.off()** è stato posto all'estremità del blocco **try** e alla fine di ogni gestore di eccezioni.

È comunque possibile che il programma sollevi un'eccezione che non viene intercettata in questo punto, nel qual caso **sw.off()** non si attiverebbe.

Tuttavia con **finally** potete disporre il codice di cleanup di un blocco **try** in un solo punto.

```
//: exceptions/WithFinally.java
// finally assicura il cleanup.

public class WithFinally {
  static Switch sw = new Switch();
  public static void main(String[] args) {
    try {
      sw.on();
      // Codice che puo' sollevare eccezioni...
      OnOffSwitch.f();
    } catch(OnOffException1 e) {
      System.out.println("OnOffException1");
    } catch(OnOffException2 e) {
      System.out.println("OnOffException2");
    } finally {
      sw.off();
    }
}
} /* Output:
on
off
*///:~
```

In questo codice **sw.off()** è stato spostato in un unico punto, dove è certo che sarà eseguito in qualsiasi caso.

Anche se l'eccezione non venisse intercettata nell'insieme corrente di istruzioni **catch**, **finally** sarebbe eseguito prima che il meccanismo di gestione delle eccezioni continui la ricerca di un gestore al successivo livello più elevato.

```
//: exceptions/AlwaysFinally.java
// finally viene sempre eseguito.
import static net.mindview.util.Print.*;

class FourException extends Exception {}

public class AlwaysFinally {
  public static void main(String[] args) {
     print("Entering first try block");
    try {
      print("Entering second try block");
      try {
        throw new FourException();
      } finally {
        print("finally in 2nd try block");
      }
    } catch(FourException e) {
      System.out.println(
        "Caught FourException in 1st try block");
    } finally {
      System.out.println("finally in 1st try block");
    }
  }
} /* Output:
Entering first try block
Entering second try block
finally in 2nd try block
Caught FourException in 1st try block
finally in 1st try block
*///:~
```

La dichiarazione **finally** viene eseguita anche nelle situazioni in cui sono implicate le dichiarazioni **break** e **continue**. Tenete presente che, se **break** e **continue** sono entrambe etichettate, **finally** elimina l'esigenza di una dichiarazione **goto**.

**Esercizio 13** (2) Modificate l'Esercizio 9 aggiungendo un'istruzione **finally**. Verificate che tale istruzione sia eseguita anche se viene sollevata una **NullPointerException**.

**Esercizio 14** (2) Dimostrate che **OnOffSwitch.java** può produrre un errore sollevando una **RuntimeException** all'interno del blocco **try**.

**Esercizio 15** (2) Dimostrate che **WithFinally.java** non produce errori sollevando una **RuntimeException** all'interno del blocco **try**.

## *Utilizzo di finally durante un return*

Dal momento che un'istruzione **finally** viene sempre eseguita, è possibile ritornare (**return**) da diversi punti di un metodo, pur garantendo che le operazioni di cleanup importanti saranno effettuate in ogni caso.

```
//: exceptions/MultipleReturns.java
import static net.mindview.util.Print.*;

public class MultipleReturns {
  public static void f(int i) {
    print("Initialization that requires cleanup");
    try {
      print("Point 1");
      if(i == 1) return;
      print("Point 2");
      if(i == 2) return;
      print("Point 3");
      if(i == 3) return;
      print("End");
      return;
    } finally {
      print("Performing cleanup");
    }
  }
  public static void main(String[] args) {
    for(int i = 1; i <= 4; i++)
      f(i);
  }
} /* Output:
Initialization that requires cleanup
Point 1
```

```
Performing cleanup
Initialization that requires cleanup
Point 1
Point 2
Performing cleanup
Initialization that requires cleanup
Point 1
Point 2
Point 3
Performing cleanup
Initialization that requires cleanup
Point 1
Point 2
Point 3
End
Performing cleanup
*///:~
```

Come vedete dall'output, non ha importanza dove si ritorna dall'interno della classe **finally**.

> **Esercizio 16** (2) Modificate **reusing/CADSystem.java** per dimostrare che da qualsiasi punto di un blocco **try-finally** si ritorni il cleanup verrà comunque effettuato correttamente.

> **Esercizio 17** (3) Modificate **polymorphism/Frog.java** utilizzando la combinazione **try-finally** per garantire il corretto cleanup, e dimostrate che questo funziona da qualsiasi punto di un blocco **try-finally** si ritorni.

## *Attenzione all'eccezione perduta*

Purtroppo l'implementazione delle eccezioni in Java presenta un inconveniente: nonostante le eccezioni indichino un evento di crisi nel vostro programma e non debbano mai essere ignorate, è possibile che un'eccezione venga semplicemente persa. Questo accade con una configurazione particolare che si serve dell'istruzione **finally**.

```
//: exceptions/LostMessage.java
// Come perdere un'eccezione.
```

```
class VeryImportantException extends Exception {
  public String toString() {
    return "A very important exception!";
  }
}
class HoHumException extends Exception {
  public String toString() {
    return "A trivial exception";
  }
}

public class LostMessage {
  void f() throws VeryImportantException {
    throw new VeryImportantException();
  }
  void dispose() throws HoHumException {
    throw new HoHumException();
  }
  public static void main(String[] args) {
    try {
      LostMessage lm = new LostMessage();
      try {
        lm.f();
      } finally {
        lm.dispose();
      }
    } catch(Exception e) {
      System.out.println(e);
    }
  }
} /* Output:
A trivial exception
*///:~
```

Come potete vedere, nell'output non vi è traccia di **VeryImportantException**, che nell'istruzione **finally** viene sostituita da **HoHumException**.

Questo è un inconveniente piuttosto grave, poiché significa che un'eccezione può essere persa, e in modi molto più sottili e difficili da rilevare rispetto all'esempio precedente.[5]

Forse una futura versione di Java porrà rimedio a questo problema; d'altra parte, tenete presente che in genere ogni metodo che solleva un'eccezione, come **dispose()** nell'esempio precedente, viene inglobato in un blocco **try-catch**.

Un modo ancora più semplice per perdere un'eccezione è fare sì che il programma ritorni (**return**) dall'interno di un'istruzione **finally**.

```
//: exceptions/ExceptionSilencer.java

public class ExceptionSilencer {
  public static void main(String[] args) {
    try {
      throw new RuntimeException();
    } finally {
      // Utilizzando 'return' all'interno del blocco finally
      // si annulla qualsiasi eccezione sia stata sollevata.
      return;
    }
  }
} ///:~
```

Se eseguite questo programma noterete che non produce output, nonostante sia sollevata un'eccezione.

**Esercizio 18** (3) Aggiungete un secondo livello di perdita di eccezione a **LostMessage.java**, in modo che anche **HoHumException** venga sostituita da una terza eccezione.

**Esercizio 19** (2) Ovviate al problema in **LostMessage.java** trasformando in guarded region la chiamata nell'istruzione **finally**.

5. C++, invece, gestisce come errore di programmazione grave la situazione in cui una seconda eccezione viene sollevata prima che quella precedente sia stata gestita.

## Limiti delle eccezioni

Quando sovrascrivete un metodo, potete sollevare soltanto le eccezioni che sono state specificate nella versione del metodo nella classe di base.

Questo è un vincolo utile, poiché significa che il codice che funziona con la classe di base funzionerà automaticamente con qualsiasi oggetto derivato da essa, incluse le eccezioni.

L'esempio seguente mostra il tipo di vincoli imposti alle eccezioni in fase di compilazione.

```
//: exceptions/StormyInning.java
// I metodi sovrascritti possono sollevare soltanto
// le eccezioni specificate
// nella rispettiva versione della classe di base,
// oppure eccezioni derivate
// da quelle della classe di base.

class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}

abstract class Inning {
  public Inning() throws BaseballException {}
  public void event() throws BaseballException {
    // Non deve sollevare nessuna eccezione
  }
  public abstract void atBat() throws Strike, Foul;
  public void walk() {} // Solleva eccezioni non controllate
}

class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}

interface Storm {
  public void event() throws RainedOut;
  public void rainHard() throws RainedOut;
}
```

```
public class StormyInning extends Inning implements Storm {
  // E' possibile aggiungere nuove eccezioni ai costruttori,
  // ma occorre gestire le eccezioni del costruttore di base:
  public StormyInning()
    throws RainedOut, BaseballException {}
  public StormyInning(String s)
    throws Foul, BaseballException {}
  // I normali metodi devono essere conformi alla classe
  // di base:
//! void walk() throws PopFoul {} // Errore di compilazione
  // L'interfaccia NON puo' aggiungere eccezioni
  // ai metodi esistenti nella classe di base:
//! public void event() throws RainedOut {}
  // Se il metodo non esiste gia' nella classe di base
  // l'eccezione e' OK:
  public void rainHard() throws RainedOut {}
  // Potete scegliere di non sollevare nessuna eccezione,
  // anche se la versione di base lo fa:
  public void event() {}
  // I metodi sovrascritti possono sollevare eccezioni
  // ereditate:
  public void atBat() throws PopFoul {}
  public static void main(String[] args) {
    try {
      StormyInning si = new StormyInning();
      si.atBat();
    } catch(PopFoul e) {
      System.out.println("Pop foul");
} catch(RainedOut e) {
      System.out.println("Rain out");
    } catch(BaseballException e) {
      System.out.println("Generic baseball exception");
    }
    // Lo strike non viene sollevato nella versione derivata.
    try {
      // Che cosa accade se si esegue un upcasting?
```

```
      Inning i = new StormyInning();
      i.atBat();
      // Bisogna intercettare le eccezioni dalla versione
      // del metodo della classe di base:
    } catch(Strike e) {
      System.out.println("Strike");
    } catch(Foul e) {
      System.out.println("Foul");
    } catch(RainedOut e) {
      System.out.println("Rain out");
    } catch(BaseballException e) {
      System.out.println("Generic baseball exception");
    }
  }
} ///:~
```

In **Inning** potete notare che sia il costruttore sia il metodo **event()** indicano che verrà sollevata un'eccezione, ma questo non accade mai. Questa tecnica è ammessa, poiché permette di costringere l'utente programmatore a intercettare tutte le eccezioni che potrebbero essere aggiunte nelle versioni sovrascritte di **event()**. Lo stesso concetto si applica ai metodi astratti, come risulta evidente in **atBat()**.

L'interfaccia **Storm** è interessante poiché contiene un metodo (**event()**) che è definito in **Inning**, e uno che non lo è: entrambi i metodi sollevano un nuovo tipo di eccezione, **RainedOut**. Quando **StormyInning** estende **Inning** e implementa **Storm**, vedrete che il metodo **event()** in **Storm** non può cambiare l'interfaccia di eccezione di **event()** in **Inning**. Anche questo ha significato perché, in caso contrario, quando lavorate con la classe di base non potrete sapere se avete intercettato l'oggetto corretto. Naturalmente non avrete problemi se un metodo descritto in un'interfaccia ma non presente nella classe di base, quale **rainHard()**, produce eccezioni.

I limiti sulle eccezioni non si applicano ai costruttori. In **StormyInning** potete osservare che un costruttore può produrre qualsiasi eccezione desideri, a prescindere da ciò che viene sollevato dal costruttore della classe di base. Tuttavia, dal momento che un costruttore di classe di base deve essere chiamato in un modo o nell'altro (nel caso di questo esempio, viene chiamato automaticamente il costruttore predefinito), il costruttore della classe derivata deve dichiarare qualsiasi eccezione del costruttore di classe di base nella propria specifica di eccezione.

Un costruttore di classe derivata non può intercettare eccezioni sollevate dal costruttore della propria classe di base.

Il motivo per cui **StormyInning.walk()** non compila è che produce un'eccezione, mentre **Inning.walk()** non lo fa. Se fosse permesso sovrascrivere **walk()** aggiungendo eccezioni, potreste scrivere codice che chiama **Inning.walk()** senza dovere gestire tutte le eccezioni; tuttavia, non appena sostituirete un oggetto di una classe derivata da **Inning** verranno sollevate le eccezioni, quindi il vostro codice produrrà errori. Invece, facendo in modo che i metodi della classe derivata siano conformi alle specifiche di eccezione dei metodi della classe di base, la sostituibilità degli oggetti viene mantenuta.

Il metodo sovrascritto **event()** mostra che una versione di classe derivata di un metodo può scegliere di non sollevare alcuna eccezione, anche se la versione della classe di base lo fa. Come si è detto, questo va benissimo perché non interferisce con il codice, che è stato scritto presupponendo che la versione della classe di base producesse eccezioni. Una logica analoga si applica ad **atBat()**, che solleva **PopFoul**, un'eccezione derivata dall'eccezione **Foul** prodotta dalla versione di classe di base di **atBat()**. In questo modo, se scrivete codice che funziona con **Inning** e chiama **atBat()** dovrete intercettare l'eccezione **Foul**; di conseguenza, poiché **PopFoul** deriva da **Foul**, il gestore di eccezioni intercetterà anche **PopFoul**.

L'ultimo punto interessante è in **main()**: qui potete vedere che, se lavorate con un solo oggetto **StormyInning**, il compilatore vi obbligherà a intercettare le sole eccezioni specifiche di questa classe, mentre se eseguite l'upcast al tipo di base il compilatore (correttamente) vi costringerà a intercettare le eccezioni del tipo di base. Tutti questi vincoli vi consentono di produrre codice per la gestione delle eccezioni decisamente più robusto.[6]

Sebbene le specifiche di eccezione vengano fatte rispettare dal compilatore anche attraverso l'ereditarietà, esse non fanno parte del tipo di metodo, che include soltanto il nome del metodo e i tipi di argomento. Di conseguenza non potete sovraccaricare i metodi basandovi sulle specifiche di eccezione. Inoltre, il fatto che nella versione di classe di base di un metodo esista una specifica di eccezione non implica che debba esistere nella versione derivata del metodo. Si tratta di un comportamento diverso dalle regole di ereditarietà, secondo la quale un metodo della classe di base deve essere presente anche nella classe derivata. In altri termini, l'"interfaccia delle specifiche di

---

6. C++ ISO ha aggiunto vincoli analoghi, secondo i quali le eccezioni dei metodi derivati devono essere uguali alle eccezioni sollevate dal metodo della classe di base, o derivare da queste ultime. Questo è un caso in cui C++ può effettivamente controllare le specifiche di eccezione in fase di compilazione.

eccezione" di un determinato metodo può essere limitata a seguito dell'ereditarietà e dell'overriding, ma certamente non ampliata: questo è l'esatto contrario della regola valida per l'interfaccia di classe in sede di ereditarietà.

**Esercizio 20** (3) Modificate **StormyInning.java** aggiungendo un tipo di eccezione **UmpireArgument** e i metodi che la sollevano, e testate la gerarchia modificata.

## Costruttori

È importante chiedersi sempre se, a seguito di un'eccezione, tutto verrà correttamente "ripulito". I problemi principali possono venire dai costruttori. Il costruttore pone l'oggetto in uno stato di sicurezza iniziale, ma potrebbe eseguire operazioni, per esempio l'apertura di un file, che non vengono sottoposte a cleanup finché l'utente programmatore non abbia terminato di utilizzare l'oggetto e chiami un metodo specifico di cleanup. Se sollevate un'eccezione dall'interno di un costruttore, questi comportamenti potrebbero non avere luogo correttamente: questo significa che dovete prestare attenzione quando scrivete il vostro costruttore.

Potreste pensare che **finally** possa rappresentare una soluzione, tuttavia **finally** esegue sempre il codice di cleanup. Se un costruttore dovesse interrompersi nel corso dell'esecuzione è possibile che non abbia creato correttamente una certa parte dell'oggetto che sarà ripulito nell'istruzione **finally**.

Nel seguente esempio viene creata una classe chiamata **InputFile** che apre un file e ne permette la lettura una riga alla volta. Il codice si serve delle classi **FileReader** e **BufferedReader** della libreria I/O standard di Java che sarà esaminata nel Capitolo 6; queste classi sono così semplici che il loro utilizzo non richiede commenti.

```
//: exceptions/InputFile.java
// Prestate attenzione alle eccezioni nei costruttori.
import java.io.*;

public class InputFile {
  private BufferedReader in;
  public InputFile(String fname) throws Exception {
    try {
in = new BufferedReader(new FileReader(fname));
      // Altro codice che potrebbe sollevare eccezioni
```

```
    } catch(FileNotFoundException e) {
      System.out.println("Could not " + fname);
      // Non era aperto, pertanto non lo chiude
      throw e;
    } catch(Exception e) {
      // Tutte le altre eccezioni devono chiuderlo
      try {
        in.close();
      } catch(IOException e2) {
        System.out.println("in.close() unsuccessful");
      }
      throw e; // Solleva nuovamente l'eccezione
    } finally {
      // Da non chiudere in questo punto!
    }
  }
  public String getLine() {
    String s;
    try {
      s = in.readLine();
    } catch(IOException e) {
      throw new RuntimeException("readLine() failed");
    }
    return s;
  }
  public void dispose() {
    try {
      in.close();
      System.out.println("dispose() successful");
    } catch(IOException e2) {
      throw new RuntimeException("in.close() failed");
    }
  }
} ///:~
```

Il costruttore di **InputFile** accetta un argomento **String** corrispondente al nome del file da aprire, e all'interno di un blocco **try** genera un **FileRea-**

**der** utilizzando il nome del file. Un **FileReader** non è particolarmente utile a meno di non utilizzarlo per creare un **BufferedReader**: uno dei vantaggi di **InputFile** è che combina queste due azioni.

Se il costruttore di **FileReader** non ha successo, solleva una **FileNotFoundException**. Questo è un tipico caso in cui non vorrete chiudere il file poiché è stato aperto correttamente. Tutte le altre istruzioni **catch** devono invece chiudere il file poiché questo è stato aperto prima che tali istruzioni fossero raggiunte. Tenete presente, naturalmente, che questo meccanismo si complica se più metodi possono produrre eccezioni di tipo **FileNotFoundException**: in tal caso, di norma occorre frazionare la funzionalità in diversi blocchi **try**. Il metodo **close()** potrebbe sollevare un'eccezione in modo da essere eseguito all'interno di un blocco **try** e intercettato anche se si trova all'interno di un'altra istruzione **catch**: questo non richiede altro che una coppia di parentesi aggiuntive per il compilatore Java. Dopo l'esecuzione delle operazioni locali l'eccezione viene prodotta di nuovo, il che è logico perché, dal momento che il costruttore corrente non ha avuto successo, non vorrete che il metodo chiamante ritenga che l'oggetto sia stato invece generato correttamente e sia valido.

In questo esempio l'istruzione **finally** non è decisamente la posizione in cui chiudere (**close()**) il file, considerato che lo chiuderebbe ogni volta che il costruttore si è concluso, mentre volete che il file rimanga aperto per tutta la durata utile dell'oggetto **InputFile**.

Il metodo **getLine()** restituisce una **String** che contiene la riga successiva del file: chiama **readLine()**, che può sollevare un'eccezione, ma tale eccezione viene intercettata in modo che **getLine()** non possa produrre eccezioni. Una delle considerazioni progettuali sulle eccezioni è capire se occorre gestire un'eccezione interamente a questo livello, gestirla in modo parziale e passare la stessa eccezione (o una diversa) a un altro livello, o semplicemente ignorarla: quest'ultima ipotesi, se appropriata, può evidentemente facilitare la scrittura del codice. In questa situazione specifica il metodo **getLine()** converte l'eccezione in una **RuntimeException** per segnalare un errore di programmazione.

Il metodo **dispose()** deve essere chiamato dall'utente programmatore quando l'oggetto **InputFile** non è più necessario: in questo modo verranno rilasciate le risorse di sistema, quali gli handle di file, utilizzate dagli oggetti **FileReader** e/o **BufferedReader**. Naturalmente non è opportuno ricorrere a **dispose()** prima di aver terminato di utilizzare l'oggetto **InputFile**. Potreste pensare di inserire tale funzionalità in un metodo **finalize()**; tuttavia, come si è detto Volume 1, Capitolo 5 non potete sempre avere la certezza che **finalize()** verrà chiamato, e se anche ne foste certi, non sapreste quando. Questo è uno degli

inconvenienti di Java: tutte le operazioni di cleanup, diverse da quelle che riguardano la memoria, non si svolgono in modo automatico, pertanto dovete informare i programmatori client che ne saranno responsabili.

Il modo più sicuro di utilizzare una classe che potrebbe sollevare un'eccezione in fase di costruzione e che richiede il cleanup è ricorrere a blocchi **try** nidificati.

```
//: exceptions/Cleanup.java
// Come garantire il cleanup accurato di una risorsa.

public class Cleanup {
  public static void main(String[] args) {
    try {
      InputFile in = new InputFile("Cleanup.java");
      try {
        String s;
        int i = 1;
        while((s = in.getLine()) != null)
          ; // Qui viene eseguita l'elaborazione di ogni
            // riga...
      } catch(Exception e) {
        System.out.println("Caught Exception in main");
        e.printStackTrace(System.out);
      } finally {
        in.dispose();
      }
    } catch(Exception e) {
      System.out.println("InputFile construction failed");
    }
  }
} /* Output:
dispose() successful
*///:~
```

Osservate con attenzione la logica di questo codice: la costruzione dell'oggetto **InputFile** avviene, in modo appropriato, all'interno del suo blocco **try**. Se questa costruzione non ha successo, si accede all'istruzione **catch** esterna e il metodo **dispose()** non viene chiamato. Invece, se la costruzione riesce, vi assicu-

rate che l'oggetto venga ripulito, pertanto subito dopo la costruzione generate un nuovo blocco **try**. L'istruzione **finally** che esegue il cleanup è associata al blocco **try** interno: in questo modo **finally** non viene eseguita se la costruzione fallisce, ed è sempre eseguita quando la costruzione termina con successo.

Questo meccanismo generale di cleanup dovrebbe essere ancora adottato se il costruttore non solleva eccezioni. La regola di base prevede di iniziare un blocco **try-finally** subito dopo avere creato un oggetto che richiede il cleanup.

```
//: exceptions/CleanupIdiom.java
// Ogni oggetto che e' soggetto a cleanup deve essere seguito
// da un blocco try-finally

class NeedsCleanup { // La costruzione non puo' fallire
  private static long counter = 1;
  private final long id = counter++;
public void dispose() {
    System.out.println("NeedsCleanup " + id + " disposed");
  }
}

class ConstructionException extends Exception {}

class NeedsCleanup2 extends NeedsCleanup {
  // La costruzione puo' fallire
  public NeedsCleanup2() throws ConstructionException {}
}

public class CleanupIdiom {
  public static void main(String[] args) {
    // Sezione 1:
    NeedsCleanup nc1 = new NeedsCleanup();
    try {
      // ...
    } finally {
      nc1.dispose();
    }
```

```
    // Sezione 2:
    // Se la costruzione non puo' fallire potete raggruppare
    // gli oggetti:
    NeedsCleanup nc2 = new NeedsCleanup();
    NeedsCleanup nc3 = new NeedsCleanup();
    try {
      // ...
    } finally {
      nc3.dispose(); // Ordine di costruzione inverso
      nc2.dispose();
    }

    // Sezione 3:
    // Se la costruzione puo' fallire dovete gestirli
    // singolarmente:
    try {
      NeedsCleanup2 nc4 = new NeedsCleanup2();
      try {
        NeedsCleanup2 nc5 = new NeedsCleanup2();
        try {
          // ...
        } finally {
          nc5.dispose();
        }
} catch(ConstructionException e) { // Costruttore nc5
        System.out.println(e);
      } finally {
        nc4.dispose();
      }
    } catch(ConstructionException e) { // Costruttore nc4
      System.out.println(e);
    }
  }
} /* Output:
NeedsCleanup 1 disposed
NeedsCleanup 3 disposed
```

```
NeedsCleanup 2 disposed
NeedsCleanup 5 disposed
NeedsCleanup 4 disposed
*///:~
```

In **main()** la sezione 1 è piuttosto semplice: a un oggetto di cui volete liberarvi fate seguire un blocco **try-finally**. Se la costruzione dell'oggetto non può fallire, non occorre alcuna istruzione **catch**. Nella sezione 2, notate che gli oggetti i cui costruttori non possono fallire sono raggruppabili sia per la costruzione sia per il cleanup.
La sezione 3 mostra come gestire gli oggetti i cui costruttori possono fallire, e che di conseguenza richiedono un'operazione di cleanup. Per far fronte in modo adeguato a questa situazione il codice si complica, poiché occorre includere ogni costruzione in un blocco **try-catch**, e ogni costruzione di oggetto deve essere seguita da un blocco **try-finally** per garantirne il cleanup.

Il fatto che la gestione delle eccezioni diventi così assurdamente complessa è un motivo sufficiente per creare costruttori "a prova di bomba", anche se questo non è sempre possibile.

Tenete presente che se **dispose()** può sollevare un'eccezione potrebbero occorrervi altri blocchi **try**. Di solito dovreste valutare con attenzione tutte le possibilità e tenere conto di ciascuna.

**Esercizio 21** (2) Dimostrate che un costruttore di classe derivata non può intercettare le eccezioni prodotte dal costruttore della sua classe di base.

**Esercizio 22** (2) Create una classe chiamata **FailingConstructor** con un costruttore che potrebbe fallire parzialmente nel processo di costruzione, sollevando un'eccezione. In **main()** scrivete il codice che vi cauteli adeguatamente da questo evento.

**Esercizio 23** (4) Integrate al codice dell'esercizio precedente una classe con un metodo **dispose()**. Modificate **FailingConstructor** in modo che il costruttore generi uno di questi oggetti "a perdere" come oggetto membro, creato il quale il costruttore potrebbe sollevare un'eccezione; poi generate un secondo oggetto membro, anch'esso "a perdere". Scrivete il codice che gestisce correttamente un eventuale guasto, e in **main()** verificate di avere tenuto conto di tutte le possibili situazioni di guasto.

**Esercizio 24** (3) Aggiungete un metodo **dispose()** alla classe **FailingConstructor** e scrivete il codice per utilizzarla correttamente.

## Corrispondenza delle eccezioni

Quando viene sollevata un'eccezione il sistema di gestione delle eccezioni cerca i gestori "più prossimi", nell'ordine in cui figurano nel codice: non appena trova quello corrispondente, l'eccezione è considerata gestita e la ricerca si interrompe.

Il criterio di corrispondenza non richiede che il riscontro tra l'eccezione e il relativo gestore sia perfetto; un oggetto di una classe derivata sarà considerato equivalente a un gestore della classe di base, come mostra il seguente esempio.

```
//: exceptions/Human.java
// Intercettazione di gerarchie di eccezioni.

class Annoyance extends Exception {}
class Sneeze extends Annoyance {}

public class Human {
  public static void main(String[] args) {
    // Intercetta il tipo esatto:
    try {
      throw new Sneeze();
    } catch(Sneeze s) {
      System.out.println("Caught Sneeze");
    } catch(Annoyance a) {
      System.out.println("Caught Annoyance");
    }
    // Intercetta il tipo di base:
    try {
      throw new Sneeze();
    } catch(Annoyance a) {
      System.out.println("Caught Annoyance");
    }
  }
} /* Output:
Caught Sneeze
Caught Annoyance
*///:~
```

L'eccezione **Sneeze** verrà intercettata dalla prima istruzione **catch** in cui trova corrispondenza, che in questo caso è la prima. Tuttavia, se eliminate la prima istruzione **catch** lasciando soltanto quella per **Annoyance**, il codice continuerà a funzionare poiché intercetterà la classe di base di **Sneeze**. In altre parole, **catch(Annoyance a)** intercetterà un'eccezione **Annoyance** o qualunque classe derivata da essa. Questa caratteristica è utile perché, se decidete di aggiungere altre eccezioni derivate a un metodo, il codice del programmatore client non dovrà essere modificato se il client intercetta le eccezioni della classe di base.

Se cercate di "mascherare" le eccezioni della classe derivata inserendo per prima l'istruzione **catch** della classe di base, come in questo esempio,

```
try {
      throw new Sneeze();
    } catch(Annoyance a) {
      // ...
    } catch(Sneeze s) {
      // ...
    }
```

il compilatore segnalerà un messaggio di errore, poiché rileverà che l'istruzione **catch** di **Sneeze** non potrà mai essere raggiunta.

> **Esercizio 25** (2) Create una gerarchia di eccezioni a tre livelli, quindi generate una classe di base **A** con un metodo che solleva un'eccezione alla base della vostra gerarchia. Poi create la classe **B** per ereditarietà da **A** e sovrascrivete il metodo in modo che produca un'eccezione al secondo livello della gerarchia. Ripetete il processo ereditando la classe **C** dalla classe **B**; in **main()** generate un oggetto della classe **C** ed eseguitene l'upcast ad **A**, quindi chiamate il metodo.

## Tecniche alternative

Un sistema di gestione delle eccezioni costituisce una sorta di via d'uscita che permette al vostro programma di interrompere l'esecuzione della normale sequenza di dichiarazioni. Questa via d'uscita è utilizzata quando si verifica una "condizione eccezionale", tale che l'esecuzione normale non sia più possibile o auspicabile: le eccezioni rappresentano condizioni che il metodo corrente non è in grado di gestire. Il motivo che ha portato allo sviluppo di sistemi per la gestione delle eccezioni è che l'approccio di con-

trollare ogni possibile condizione di errore prodotta da qualsiasi chiamata di funzione era troppo oneroso, e i programmatori semplicemente non ne tenevano conto; di conseguenza iniziavano a ignorare gli errori. È interessante osservare come il problema della praticità nella gestione degli errori da parte dei programmatori sia stato la motivazione principale che ha dato origine alle eccezioni.

Una delle regole fondamentali nella gestione delle eccezioni indica di non intercettare un'eccezione a meno di non sapere come governarla. In effetti, uno degli obiettivi importanti in questa gestione è spostare il codice di gestione dell'errore in avanti rispetto al punto in cui si verifica l'errore. Questo vi consente di concentrarvi su ciò che ritenete opportuno eseguire in una sezione del codice, e su come occuparvi dei problemi in una diversa sezione. In questo modo il vostro flusso di codice principale non sarà disseminato di frammenti di logica per la gestione degli errori, e risulterà molto più facile da comprendere e mantenere. La gestione delle eccezioni presenta anche il vantaggio di ridurre la quantità di codice, facendo sì che un gestore si occupi di diversi punti di potenziali errori.

Le eccezioni controllate complicano in qualche modo questa situazione, perché vi obbligano ad aggiungere istruzioni **catch** nei punti in cui potreste non essere pronti per gestire un errore. Questo provoca il cosiddetto problema "nocivo se ingerito".

```
try {
  // ... per eseguire qualcosa di utile
} catch(ObligatoryException e) {} // Gulp!
```

I programmatori (incluso lo stesso autore, nella prima edizione di questo libro) adotterebbero l'approccio più semplice e "inghiottirebbero" l'eccezione, spesso senza riflettere sulle conseguenze: tuttavia, una volta che lo si è fatto, il compilatore si considera soddisfatto, pertanto, a meno di non rivedere e correggere il codice, l'eccezione sarà persa. L'eccezione si verifica ma sparisce completamente dopo essere stata "inghiottita". Poiché il compilatore vi costringe a scrivere subito il codice per gestire l'eccezione, questa sembra essere la soluzione più semplice, benché sia probabilmente l'errore peggiore che possiate commettere.

Dopo essersi reso conto di avere commesso un tale errore, nella seconda edizione del libro l'autore ha "risolto" il problema visualizzando lo stack trace all'interno del gestore, tecnica che si riscontra ancora, appropriatamente, in alcuni esempi in questo capitolo. Sebbene questo meccanismo sia utile per tenere traccia del comportamento delle eccezioni, continua a indicare che ancora non

si sa come gestire l'eccezione in quel punto del codice. In questo paragrafo vedrete alcuni problemi e complicazioni che possono sorgere a fronte delle eccezioni controllate, e le possibilità che avete a disposizione per gestirle.

Questo argomento sembra semplice, mentre in realtà non è soltanto complesso ma anche in un certo senso instabile. C'è chi sostiene che sia preferibile gestire l'eccezione appena si verifica, e chi invece reputa più idoneo rigenerarla fino a quando non si sa con precisione cosa fare. L'autore ritiene che il motivo per sostenere una di queste posizioni sia il distinto beneficio percepito nel passaggio da un linguaggio "sciolto" come il vecchio C pre-ANSI a uno più forte e statico, ovvero controllato al momento della compilazione, quale C++ o Java. Quando procedete a questa transizione, come ha fatto l'autore, i vantaggi risultano così notevoli da fare sempre apparire il controllo statico come la risposta migliore alla gran parte dei problemi.

## Storia

La gestione delle eccezioni ha avuto origine in linguaggi quali PL/1 e MESA, e più tardi ha fatto la sua apparizione in CLU, Smalltalk, Modula-3, Ada, Eiffel, C++, Python, Java e i linguaggi successivi a Java, Ruby e C#. Lo schema progettuale di Java è analogo a quello di C++, tranne nei punti in cui i progettisti Java hanno ritenuto che le tecniche di C++ avrebbero causato problemi.

Per fornire ai programmatori una struttura che fossero più propensi a utilizzare per la gestione degli errori e il ripristino, la gestione delle eccezioni in C++ è stata aggiunta con un certo ritardo, nel processo di normalizzazione promosso da Bjarne Stroustrup, l'autore originale di questo linguaggio. Il modello per le eccezioni di C++ proviene principalmente da CLU; tuttavia a quel tempo esistevano altri linguaggi che supportavano la gestione delle eccezioni: Ada e Smalltalk, dotati di eccezioni ma non di specifiche di eccezione, e Modula-3, che le includeva entrambe.

Nella loro fondamentale trattazione sull'argomento, Liskov e Snyder osservano che uno dei principali difetti di linguaggi come il C, che segnalano gli errori in modo transitorio, è che:

> *"...ogni chiamata deve essere seguita da un test condizionale per determinare quale sia il risultato. Questa necessità conduce alla realizzazione di programmi difficili da leggere e probabilmente anche inefficienti, e scoraggia i programmatori dal segnalare e gestire le eccezioni."*[7]

---

7. Barbara Liskov e Alan Snyder, "Exception Handling in CLU", *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 6, Novembre 1979. Questo documento non è disponibile

Di conseguenza una delle motivazioni originali per la gestione delle eccezioni era impedire questo requisito, tuttavia con le eccezioni controllate di Java questo genere di codice si riscontra comunemente. Liskov e Snyder continuano con:

> *"...richiedere che il codice di un gestore sia collegato alla chiamata che solleva l'eccezione porterebbe alla creazione di programmi illeggibili, in cui le espressioni sarebbero interrotte dai vari gestori."*

Attenendosi al "metodo CLU" nella progettazione delle eccezioni di C++, Stroustrup ha quindi evidenziato che l'obiettivo era ridurre la quantità di codice richiesta per il ripristino dagli errori. L'autore è convinto che Stroustrup avesse rilevato che i programmatori non scrivevano codice per la gestione degli errori in C, poiché la quantità e la disposizione di questo codice erano tali da scoraggiare e fare perdere la concentrazione a chiunque; di conseguenza i programmatori erano soliti adottare il "metodo C", ignorando gli errori nel codice e ricorrendo al debugging per individuare i problemi. Per adottare le eccezioni, i programmatori C avrebbero dovuto essere convinti ad aggiungere codice che non erano soliti scrivere. Quindi, indirizzandoli verso una tecnica migliore per gestire gli errori, la quantità di codice che avrebbero dovuto "aggiungere" non sarebbe stata eccessiva. È importante tenere presente questo obiettivo quando si considerano gli effetti delle eccezioni controllate in Java.

C++ ha introdotto un concetto supplementare derivato da CLU: la specifica di eccezione, per dichiarare programmaticamente nella segnatura del metodo le eccezioni che potrebbero derivare dalla chiamata di quel metodo. In realtà, le specifiche di eccezione hanno due scopi. Possono volere dire "Ho originato questa eccezione nel mio codice: gestitela", ma anche significare "Ignoro questa eccezione che potrebbe verificarsi nel mio codice: gestitela". Analizzando i meccanismi e la sintassi delle eccezioni, fino a questo punto avete visto essenzialmente il componente "gestitela", ma in questa sede è interessante in particolare il fatto che spesso le eccezioni vengono ignorate, e questo è quanto possono dichiarare le specifiche di eccezione.

In C++ le specifiche di eccezione non fanno parte delle informazioni di tipo di una funzione. L'unico controllo eseguito dal compilatore consiste nell'accertarsi che le specifiche vengano utilizzate in modo coerente: per esempio, se una funzione o un metodo sollevano eccezioni, allora anche le versioni sovraccariche o derivate dovranno produrre le stesse eccezio-

---

su Internet, ma soltanto in formato cartaceo, pertanto dovrete rivolgervi a una biblioteca per consultarlo.

ni. A differenza di Java, tuttavia, C++ non esegue alcun controllo in fase di compilazione per determinare se la funzione o il metodo produrranno effettivamente quell'eccezione, né se le specifiche di eccezione sono complete, vale a dire se descrivono esattamente tutte le eccezioni che possono essere sollevate. Quella convalida viene eseguita, ma soltanto al momento dell'esecuzione. Se il codice produce un'eccezione che viola le specifiche di eccezione, il programma C++ chiamerà la funzione di libreria standard **unexpected()**.

È interessante notare che, a causa dell'utilizzo delle maschere (*template*), le specifiche di eccezione non vengono utilizzate nella libreria standard di C++. In Java, di contro, esistono limitazioni sul modo di utilizzare i generici con le specifiche di eccezione.

## Prospettive

In primo luogo è opportuno notare che Java ha inventato l'eccezione controllata, chiaramente ispirata alle specifiche di eccezione di C++ e al fatto che i programmatori di C++ di solito non se ne curano. Tuttavia si è trattato di un esperimento che nessun linguaggio successivo ha scelto di ripetere.

Secondariamente le eccezioni controllate sembrano essere una "buona cosa ovvia", quando sono presenti negli esempi introduttivi e nei piccoli programmi. Le difficoltà cominciano a comparire via via che i programmi aumentano di dimensioni. Naturalmente il problema delle dimensioni non si verifica da un momento all'altro, ma si "insinua" in modo graduale. I linguaggi che potrebbero non essere adatti per progetti su larga scala vengono impiegati in quelli di piccole dimensioni; questi progetti si sviluppano e a un certo punto ci si rende conto che le cose sono passate dallo stadio di "gestibili" a quello di "difficili". Questo è quanto l'autore ritiene possa accadere in seguito a controlli eccessivi sul tipo, in particolare con le eccezioni controllate.

Le dimensioni del programma sembrano essere un problema significativo, poiché generalmente la maggior parte delle discussioni teoriche si serve di piccoli programmi come dimostrazioni. Uno dei progettisti di C# ha osservato che:

> *"L'analisi di piccoli programmi porta alla conclusione che la richiesta di specifiche di eccezione potrebbe migliorare sia il rendimento dello sviluppatore sia la qualità di codice, tuttavia l'esperienza con grandi progetti software evidenzia un risultato diverso: calo di produttività e poco o nessun miglioramento della qualità del codice."*[8]

8. *http://discuss.develop.com/archives/wa.exe?A2=ind0011A&L=DOTNET&P=R32820*

Con riferimento alle eccezioni non intercettate, i creatori di CLU hanno dichiarato:

> *"Abbiamo ritenuto che non fosse realistico richiedere al programmatore di fornire i gestori in situazioni in cui non potesse essere eseguita nessuna azione utile."*[9]

Nell'illustrare i motivi per cui una dichiarazione di funzione priva di specifiche può sollevare qualche eccezione, piuttosto che nessuna, Stroustrup afferma:

> *"Tuttavia questo richiederebbe una specifica di eccezione praticamente per ogni funzione, sarebbe motivo ricorrente di ricompilazione e limiterebbe la cooperazione con il software scritto in altri linguaggi. Questo invoglierebbe i programmatori a "sconvolgere" i meccanismi di gestione delle eccezioni, scrivendo codice illegittimo nell'intento di sopprimerle. Fornirebbe un falso senso di sicurezza a coloro che non sono riusciti a rilevare l'eccezione."*[10]

Questo stesso comportamento (lo "sconvolgimento" delle eccezioni) si riscontra nelle eccezioni controllate di Java.

Martin Fowler, autore di *UML Distilled, Refactoring, and Analysis Patterns*, ha scritto questa nota all'autore:

> *"...dopo tutto penso che le eccezioni siano una buona cosa, ma le eccezioni controllate di Java creano più difficoltà di quante ne risolvano."*

Oggi l'autore ritiene che il punto importante di Java sia stato l'avere unificato il modello di segnalazione degli errori, in modo che tutti vengano segnalati per mezzo di eccezioni. Questo non è accaduto con C++: per motivi di retrocompatibilità con C il vecchio modello che si limitava a ignorare gli errori era ancora valido. Ma se segnalate costantemente le eccezioni, queste potranno essere utilizzate nel modo che riterrete opportuno; in caso contrario, si propagheranno al successivo livello più elevato, ossia la console o un altro programma contenitore. Quando Java ha modificato il modello di C++ in modo che le eccezioni fossero l'unica modalità per segnalare gli errori, l'applicazione supplementare delle eccezioni controllate avrebbe potuto diventare non indispensabile.

---

9. *Barbara Liskov e Alan Snyder, ibidem.*
10. Bjarne Stroustrup, *The C++ Programming Language*, 3ª edizione, Addison-Wesley, 1997, p. 376.

In passato, l'autore è stato un fervente sostenitore del fatto che sia le eccezioni controllate sia il controllo di tipo statico sono requisiti essenziali per lo sviluppo di un programma robusto. Tuttavia, l'esperienza sia diretta sia indiretta con linguaggi più dinamici che statici lo ha portato a pensare che i benefici più notevoli in realtà derivino da:[11]

1. un modello unificato per la segnalazione degli errori tramite eccezioni, a prescindere se il programmatore è costretto dal compilatore a gestirlo;
2. il controllo del tipo, a prescindere da quando viene svolto. In pratica, purché venga implementato l'utilizzo corretto di un tipo, spesso non ha importanza se il controllo avviene al momento della compilazione o dell'esecuzione.

Oltre a questo, la riduzione dei vincoli imposti al programmatore al momento della compilazione comporta vantaggi notevoli in termini di produttività. In realtà, il meccanismo di riflessione e i generici sono richiesti per compensare la natura "sovralimitativa" della cosiddetta tipizzazione in sede di compilazione (*static typing*, contrapposta alla tipizzazione in fase di esecuzione o *dynamic typing*, tipica di Smalltalk), come vedrete in alcuni esempi di questo libro.

Qualche lettore ha già fatto osservare che le affermazioni dell'autore in questo contesto sono "blasfeme", e che pronunciando queste parole la sua reputazione sarà messa a rischio, le civiltà saranno distrutte e una notevole percentuale di progetti software non potrà essere completata. La convinzione che il compilatore possa "salvare" i progetti evidenziando gli errori in fase di compilazione rimane radicata, tuttavia è ancora più importante rendersi conto dei limiti operativi del compilatore; nel supplemento che troverete all'indirizzo *http://mindview.net/Books/BetterJava*, l'autore mette in risalto il valore di un processo di build automatico e di test unitari, che vi garantiranno molta più potenza di quella che otterreste trasformando tutto in un errore di sintassi. Vale la pena tenere presente che:

> *"Un buon linguaggio di programmazione è quello che aiuta i programmatori a scrivere buoni programmi. Nessun linguaggio di programmazione impedirà che gli sviluppatori scrivano cattivi programmi."*[12]

In ogni caso sembra improbabile che le eccezioni controllate vengano mai rimosse da Java. Sarebbe un cambiamento troppo radicale nel linguaggio e

11. Indirettamente con Smalltalk, tramite conversazioni con molti programmatori esperti di questo linguaggio, e direttamente con Python (*www.python.org*).
12. Kees Koster, progettista del linguaggio CDL, citato da Bertrand Meyer, progettista del linguaggio Eiffel.

i fautori delle eccezioni controllate nell'ambito di Sun sembrano essere una lobby molto potente. Sun ha una lunga storia e politiche di totale retro-compatibilità: per darvi un'idea di quanto questo sia importante per l'azienda, sappiate che praticamente tutti i software Sun funzionano su qualsiasi hardware Sun, per quanto vecchio sia. Tuttavia, se ritenete che alcune eccezioni controllate vi siano d'intralcio, e soprattutto se vi trovate costretti a intercettare le eccezioni ma non sapete come gestirle, troverete utili alcune alternative.

## *Passaggio delle eccezioni alla console*

Nei programmi semplici, come molti di quelli presentati in questo libro, il modo più facile per conservare le eccezioni senza dover scrivere una quantità elevata di codice consiste nel passarli da **main()** alla console. Per esempio, se volete aprire un file in lettura, come vedrete meglio nel Capitolo 6, dovrete aprire e chiudere un **FileInputStream**, che solleva eccezioni. Per un programma semplice potete adottare questo accorgimento, che troverete ripetuto in numerosi esempi di questo libro.

```
//: exceptions/MainException.java
import java.io.*;

public class MainException {
  // Passa tutte le eccezioni alla console:
  public static void main(String[] args) throws Exception {
    // Apre il file:
    FileInputStream file =
      new FileInputStream("MainException.java");
    // Usa il file...
    // Chiude il file:
    file.close();
  }
} ///:~
```

Notate che anche il metodo **main()** può avere una specifica di eccezione; in questo caso il tipo è **Exception**, la classe radice di tutte le eccezioni controllate. Passando le eccezioni alla console siete dispensati dallo scrivere i costrutti **try-catch** nel corpo del metodo **main()**. Purtroppo, l'I/O sui file è notevol-

mente più complesso di quanto possa risultare da questo esempio, quindi non entusiasmatevi troppo almeno finché non avrete letto il Capitolo 6.

**Esercizio 26** (1) Cambiate la stringa che contiene il nome di file in **MainException.java**, specificando il nome di un file inesistente, poi eseguite il programma e prendete nota dei risultati.

## *Conversione delle eccezioni da controllate a non controllate*

Sollevare un'eccezione da **main()** è pratico quando scrivete semplici programmi per vostro uso personale, ma non è utile in generale. Il vero problema nasce quando state scrivendo il corpo di un comune metodo e ne chiamate un altro, rendendovi conto che in quel punto non avete alcuna idea di che cosa fare dell'eccezione, che non volete perdere né visualizzare sotto forma di un banale messaggio di errore. Con le eccezioni concatenate, una soluzione nuova e semplice evita questa incombenza. È sufficiente "inglobare" un'eccezione controllata in una **RuntimeException**, passandole il costruttore di **RuntimeException**, come nell'esempio seguente:

```
try {
  // ... per fare qualcosa di utile
} catch(IDontKnowWhatToDoWithThisCheckedException e) {
  throw new RuntimeException(e);
}
```

Questa sembra una soluzione ideale per "disattivare" l'eccezione controllata: non la "inghiottite" né dovete inserirla tra le specifiche delle eccezioni del vostro metodo, ma grazie al concatenamento non perdete alcuna informazione dell'eccezione originale.

Questa tecnica fornisce la possibilità di ignorare l'eccezione pur permettendole di rimanere in primo piano nello stack delle chiamate, senza essere obbligati a scrivere istruzioni **try-catch** e/o specifiche di eccezione. In ogni caso potete ancora intercettare e gestire l'eccezione specifica tramite **getCause()**, come nell'esempio seguente.

```
//: exceptions/TurnOffChecking.java
// "Disattivazione" delle eccezioni controllate.
import java.io.*;
import static net.mindview.util.Print.*;
```

```
class WrapCheckedException {
  void throwRuntimeException(int type) {
    try {
      switch(type) {
        case 0: throw new FileNotFoundException();
        case 1: throw new IOException();
        case 2: throw new RuntimeException("Where am I?");
        default: return;
      }
    } catch(Exception e) { // Adattamento alle eccezioni
                           // non controllate:
      throw new RuntimeException(e);
    }
  }
}

class SomeOtherException extends Exception {}

public class TurnOffChecking {
public static void main(String[] args) {
    WrapCheckedException wce = new WrapCheckedException();
    // E' possibile chiamare throwRuntimeException()
    // senza un blocco try e permettere a RuntimeExceptions
    // di lasciare il metodo:
    wce.throwRuntimeException(3);
    // Oppure potete scegliere di intercettare le eccezioni:
    for(int i = 0; i < 4; i++)
      try {
        if(i < 3)
          wce.throwRuntimeException(i);
        else
          throw new SomeOtherException();
      } catch(SomeOtherException e) {
          print("SomeOtherException: " + e);
      } catch(RuntimeException re) {
        try {
```

```
          throw re.getCause();
        } catch(FileNotFoundException e) {
          print("FileNotFoundException: " + e);
        } catch(IOException e) {
          print("IOException: " + e);
        } catch(Throwable e) {
          print("Throwable: " + e);
        }
      }
  }
} /* Output:
FileNotFoundException: java.io.FileNotFoundException
IOException: java.io.IOException
Throwable: java.lang.RuntimeException: Where am I?
SomeOtherException: SomeOtherException
*///:~
```

Il metodo **WrapCheckedException.throwRuntimeException()** contiene il codice per generare diversi tipi di eccezioni. Queste vengono intercettate e inserite in oggetti **RuntimeException**, così da diventare la "causa" di queste eccezioni.

In **TurnOffChecking** potete notare che è possibile chiamare **throwRuntimeException()** senza la presenza di un blocco **try**, poiché il metodo non produce nessuna eccezione controllata. Tuttavia quando siete pronti per intercettare le eccezioni avete ancora la capacità di farlo, inserendo il codice opportuno in un blocco **try**. Iniziate intercettando tutte le eccezioni che siete certi possano emergere dal codice nel vostro blocco **try**: in questo esempio, **SomeOtherException** viene intercettata per prima. Per ultima intercettate **RuntimeException** e sollevate con **throw** il risultato di **getCause()**, vale a dire l'eccezione inglobata. Questa tecnica estrae le eccezioni originali, che possono essere poi gestite nelle opportune istruzioni **catch**.

Nel prosieguo di questo manuale la tecnica di inglobare un'eccezione controllata in un oggetto **RuntimeException** è stata applicata quando lo si è ritenuto conveniente. Un'altra soluzione consiste nel creare le vostre sottoclassi di **RuntimeException**: in questo modo non sarà necessario intercettarle, ma il lettore che lo ritenesse opportuno potrà farlo.

**Esercizio 27** (1) Modificate l'Esercizio 3 per convertire l'eccezione in una **RuntimeException**.

**Esercizio 28** (1) Modificate l'Esercizio 4 in modo che la classe di eccezione personalizzata erediti da **RuntimeException** e dimostrate che il compilatore consente di omettere il blocco **try**.

**Esercizio 29** (1) Modificate tutti i tipi di eccezione in **StormyInning.java** in modo che estendano **RuntimeException** e dimostrate che non sono necessarie né le specifiche di eccezione né i blocchi **try**. Eliminate poi i commenti "**//!**" e dimostrate che i metodi possono essere compilati senza specifiche.

**Esercizio 30** (2) Modificate **Human.java** in modo che le eccezioni ereditino da **RuntimeException**. Poi modificate il metodo **main()** affinché gestisca i vari tipi di eccezioni con la tecnica utilizzata in **TurnOffChecking.java.**

## Guida di riferimento alle eccezioni

Utilizzate le eccezioni per:

1. gestire i problemi al livello opportuno: evitate di intercettare le eccezioni a meno che non sappiate come gestirle;
2. correggere i problemi e richiamare il metodo che ha provocato l'eccezione;
3. ripristinare l'esecuzione e proseguire senza chiamare nuovamente il metodo;
4. ottenere risultati alternativi rispetto a quanto il metodo avrebbe dovuto produrre;
5. fare quanto è possibile nel contesto corrente, e sollevare di nuovo la stessa eccezione a un contesto più elevato;
6. fare quanto è possibile nel contesto corrente, e sollevare una diversa eccezione a un contesto più elevato;
7. terminare il programma;
8. semplificare il codice (se il vostro schema di gestione delle eccezioni è troppo complesso, può essere difficile e noioso da utilizzare);
9. rendere più sicuri i vostri programmi e le vostre librerie. Questo è un investimento a breve termine in un'ottica di debugging, ma a lungo termine dal punto di vista della robustezza delle applicazioni.

## Riepilogo

Le eccezioni sono così integrate nella programmazione Java che non è possibile progredire nello studio di questo linguaggio senza conoscerne il funzionamento. Per questo motivo si è ritenuto essenziale introdurre l'argomento in questo punto del manuale: esistono molte librerie, come la già citata I/O, che non potrete utilizzare senza gestire anche le eccezioni.

Uno dei vantaggi della gestione delle eccezioni è che vi consente di concentrarvi in un solo punto sul problema da risolvere, per poi gestire gli errori provenienti da quel codice in un punto diverso. Benché le eccezioni siano generalmente presentate come strumenti che permettono di segnalare gli errori in fase di esecuzione e di ripristinare l'esecuzione stessa, l'autore si chiede quanto spesso l'"aspetto recupero" sia implementato o anche possibile: la sua opinione è che si tratti di meno del 10% dei casi, e anche in queste situazioni probabilmente sarebbe sufficiente il semplice ripristino dello stack a uno stato stabile noto, invece di eseguire effettivamente un ripristino di qualsiasi genere. Che questo sia vero o meno, l'autore ritiene che il reale valore delle eccezioni risieda nella funzionalità di "notifica". Il fatto che Java insista sul fatto che tutti gli errori vengano segnalati sotto forma di eccezioni è ciò che gli conferisce un notevole vantaggio rispetto a linguaggi come C++, che permettono di segnalare gli errori con modalità variabili o di non segnalarli affatto. Un sistema coerente di gestione degli errori implica che non dovrete mai più chiedervi, per ogni porzione di codice che scriverete, se qualche errore possa esservi sfuggito (a meno di non decidere di "inghiottire" le eccezioni, naturalmente!).

Come vedrete nei prossimi capitoli, se accantonate questo argomento, anche se gestite gli errori sollevando una **RuntimeException**, avrete l'opportunità di concentrare i vostri sforzi di progettazione e implementazione su questioni più interessanti e stimolanti.

> *La soluzione degli esercizi è disponibile nel documento* The Thinking in Java Annotated Solution Guide, *in vendita all'indirizzo* www.mindview.net.

# Capitolo 2

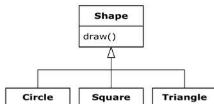
# Informazioni sui tipi

La funzionalità RTTI (*RunTime Type Information*), o *identificazione adinamica dei tipi*, consente di scoprire e utilizzare le informazioni sui tipi mentre un programma è in funzione, liberandovi dal vincolo di operare in modo orientato ai tipi soltanto in fase di compilazione, e permette di realizzare programmi molto potenti. La necessità di RTTI è fonte di molti argomenti di progettazione OOP assai interessanti, talvolta imbarazzanti, e di considerazioni fondamentali sul modo in cui dovreste strutturare i vostri programmi.

Questo capitolo tratta le tecniche Java mediante le quali potete ottenere informazioni sugli oggetti e le classi in fase di esecuzione. Tale funzionalità assume due forme distinte: l'RTTI "tradizionale", che presuppone la disponibilità di tutti i tipi in fase di compilazione, e il meccanismo detto di *riflessione*, che permette di scoprire e utilizzare le informazioni sulle classi unicamente al momento dell'esecuzione.

## L'esigenza di RTTI

Considerate l'esempio ormai familiare di una gerarchia di classi che utilizza il polimorfismo. Il tipo generico è la classe di base **Shape** e i tipi derivati specifici sono **Circle**, **Square** e **Triangle**.




Questo è un tipico diagramma di una gerarchia di classi, con la classe di base in alto e le classi derivate che si sviluppano verso il basso. L'obiettivo tipico della programmazione orientata agli oggetti consiste nel manipolare i riferimenti al tipo di base (**Shape**), in modo da poter estendere il programma aggiungendo una nuova classe, come **Rhomboid**, derivata da **Shape**, senza influenzare il codice esistente. In questo esempio, il metodo collegato dinamicamente all'interfaccia di **Shape** è **draw()**, pertanto l'intenzione è fare in modo che il programmatore client chiami il metodo **draw()** tramite un riferimento generico a **Shape**. In tutte le classi derivate, il metodo **draw()** è sovrascritto, ed essendo collegato dinamicamente produrrà il comportamento corretto anche se chiamato tramite un riferimento a un oggetto generico **Shape**. Questo, in sintesi, è il polimorfismo.

Pertanto, generalmente creerete un oggetto specifico (**Circle**, **Square** o **Triangle**), ne eseguirete l'upcast a **Shape** per "dimenticare" il tipo specifico dell'oggetto, e utilizzerete questo riferimento anonimo a **Shape** nel resto del programma.

La gerarchia **Shape** può essere codificata come segue.

```
//: typeinfo/Shapes.java
import java.util.*;

abstract class Shape {
  void draw() { System.out.println(this + ".draw()"); }
  abstract public String toString();
}

class Circle extends Shape {
  public String toString() { return "Circle"; }
}
```

```
class Square extends Shape {
  public String toString() { return "Square"; }
}

class Triangle extends Shape {
  public String toString() { return "Triangle"; }
}

public class Shapes {
  public static void main(String[] args) {
    List<Shape> shapeList = Arrays.asList(
      new Circle(), new Square(), new Triangle()
    );
    for(Shape shape : shapeList)
      shape.draw();
  }
} /* Output:
Circle.draw()
Square.draw()
Triangle.draw()
*///:~
```

La classe di base contiene un metodo **draw()** che utilizza indirettamente il metodo **toString()** per visualizzare un identificativo della classe, passando **this** a **System.out.println()**: notate che **toString()** è dichiarato **abstract** per costringere le classi che ereditano a sovrascriverlo e per impedire che venga istanziato un oggetto **Shape**. Se un oggetto figura in un'espressione di concatenamento di stringhe che coinvolge l'operatore "**+**" e oggetti **String**, viene chiamato automaticamente il metodo **toString()** per produrre una rappresentazione **String** dell'oggetto. Ciascuna delle classi derivate sovrascrive il metodo **toString()** originario della classe **Object**, e per questo il metodo **draw()** si comporta in modo polimorfo, visualizzando dati diversi in base alla situazione.

In questo esempio l'upcast avviene quando la **Shape** viene inserita in **List<Shape>**. Durante l'operazione di upcasting, la nozione che gli oggetti sono tipi specifici di **Shape** viene persa: per l'array si tratta semplicemente di oggetti **Shape**.

Nel momento in cui prelevate un elemento dall'array, il contenitore, che in realtà conserva qualsiasi oggetto sotto forma di **Object**, esegue automaticamente l'upcast di nuovo a una **Shape**. Questa è la forma più elementare di RTTI, in cui tutte le conversioni vengono controllate durante l'esecuzione al fine di garantirne la correttezza. In sostanza, il significato di RTTI è "identificazione del tipo di oggetto in fase di esecuzione".

In questo caso la conversione RTTI è soltanto parziale: **Object** è convertito in una **Shape** e non nei tipi originari **Circle**, **Square** o **Triangle**, dal momento che tutto ciò che sapete a questo punto è che **List<Shape>** contiene oggetti **Shape**. Al momento della compilazione il rispetto di queste regole è garantito dal contenitore e dal motore Java, tuttavia in fase di esecuzione è la conversione che se ne occupa.

A questo punto la funzionalità di polimorfismo assume il controllo, e il codice esatto da eseguire per l'oggetto **Shape** viene determinato valutando se il riferimento è di tipo **Circle**, **Square** o **Triangle**. In generale questo è il comportamento che ci si aspetta; in effetti, si vuole che la maggior parte del codice "sappia" il meno possibile sui tipi specifici degli oggetti, e si occupi principalmente della rappresentazione generale di una famiglia degli oggetti, in questo caso **Shape**. In quest'ottica il vostro codice sarà più facile da scrivere, leggere e sottoporre a manutenzione e i vostri progetti saranno più facili da implementare, comprendere e modificare. Il polimorfismo, quindi, deve essere considerato l'obiettivo generale della programmazione orientata agli oggetti.

Vi chiederete, tuttavia, che cosa succederebbe con un problema di programmazione particolare, più facile da risolvere conoscendo il tipo esatto di un riferimento generico. Supponete, per esempio, di volere permettere all'utente di evidenziare tutte le figure di un tipo specifico, assegnando loro un colore particolare: in questo modo, per esempio, potrebbe trovare tutti i triangoli sullo schermo semplicemente evidenziandoli. Oppure, immaginate che il vostro metodo debba "ruotare" un elenco di forme; non avendo senso ruotare un cerchio, l'oggetto **Circle** potrebbe essere ignorato. RTTI permette di chiedere a un riferimento **Shape** il tipo esatto cui esso fa riferimento, per selezionare e isolare i casi particolari.

## L'oggetto Class

Per comprendere come funziona il meccanismo RTTI in Java dovete innanzitutto sapere come le informazioni sui tipi vengono rappresentate in fase di esecuzione. Questa operazione viene eseguita mediante un tipo di oggetto speciale chiamato oggetto **Class** (***Class** object*), che contiene le informazioni relative alla classe. In effetti, l'oggetto **Class** viene impiegato per creare tutti

gli oggetti "normali" di una determinata classe. Java esegue l'identificazione dinamica dei tipi utilizzando l'oggetto **Class**, anche quando si eseguono operazioni di casting. Ma il **Class** della classe offre anche altri modi per utilizzare l'RTTI.

Esiste un oggetto **Class** per ogni classe che compone il vostro programma: in pratica, ogni volta che scrivete e compilate una nuova classe viene generato un oggetto **Class**, che è archiviato in un file con il nome della classe e l'estensione **.class**. Per creare un oggetto di questa classe, la macchina virtuale JVM che esegue il programma si serve di un sottosistema chiamato *class loader* o caricatore di classe.

Questo sottosistema, in realtà, può comprendere una catena di caricatori di classi, tuttavia esiste un solo caricatore di classi principale (*primordial class loader*) che fa parte dell'implementazione della JVM. Il caricatore di classi principale carica le cosiddette *trusted classes* (classi fidate), incluse quelle delle API Java, di norma dal disco fisso locale. Solitamente non occorre che la catena includa altri caricatori, ma se avete esigenze specifiche, per esempio il caricamento di classi che supportano applicazioni per server web o il download di classi dalla rete, può essere opportuno integrare caricatori supplementari.

Al loro primo utilizzo tutte le classi vengono caricate dinamicamente nella JVM; questo avviene quando il programma crea il primo riferimento a un membro **static** della classe corrente.

Anche se non si specifica la parola chiave **static** per i costruttori, essi sono comunque metodi **static** della classe: per questo motivo la creazione di un nuovo oggetto di una classe tramite l'operatore **new** conta come riferimento a un membro **static** della classe.

Pertanto, un programma Java non viene completamente caricato prima del suo avvio, poiché alcuni dei suoi componenti sono caricati quando necessario. Questo comportamento è diverso rispetto a quello di molti linguaggi tradizionali. Il caricamento dinamico permette di produrre comportamenti che sono difficili o impossibili da duplicare in un linguaggio caricato staticamente, come C++.

In primo luogo, il caricatore di classe controlla se l'oggetto **Class** relativo al tipo corrente è già caricato; qualora non lo sia, il caricatore della classe predefinito troverà i file **.class** registrati con quel nome: tenete presente che un caricatore di classe aggiuntivo potrebbe però cercare i bytecode in un database, per esempio. Quando il codice della classe è caricato Java lo verifica, per assicurarsi che non sia danneggiato e che non includa istruzioni potenzialmente pericolose: questa è una delle linee di difesa che Java offre in termini di sicurezza.

Quando l'oggetto **Class** del tipo in questione è in memoria viene utilizzato per generare tutti gli oggetti di questo tipo. L'esempio seguente dimostra questo comportamento.

```
//: typeinfo/SweetShop.java
// Verifica del funzionamento del class loader
import static net.mindview.util.Print.*;

class Candy {
  static { print("Loading Candy"); }
}

class Gum {
  static { print("Loading Gum"); }
}

class Cookie {
  static { print("Loading Cookie"); }
}

public class SweetShop {
  public static void main(String[] args) {
    print("inside main");
    new Candy();
    print("After creating Candy");
    try {
      Class.forName("Gum");
    } catch(ClassNotFoundException e) {
      print("Couldn't find Gum");
    }
    print("After Class.forName(\"Gum\")");
    new Cookie();
    print("After creating Cookie");
  }
} /* Output:
inside main
Loading Candy
```

```
After creating Candy
Loading Gum
After Class.forName("Gum")
Loading Cookie
After creating Cookie
*///:~
```

Le tre classi **Candy**, **Gum** e **Cookie** hanno una clausola **static** che viene eseguita al primo caricamento della classe; per ognuna vengono visualizzate informazioni che indicano quando avviene il caricamento. In **main**() la creazione degli oggetti è intervallata da varie istruzioni di visualizzazione, per consentirvi di rilevare il caricamento delle classi.

Come potete notare, ogni oggetto **Class** viene caricato soltanto al momento opportuno, e l'inizializzazione **static** viene eseguita al caricamento della classe.

Una riga di codice particolarmente interessante è la seguente:

```
Class.forName("Gum");
```

Tutti gli oggetti **Class** appartengono alla classe **Class**. Un oggetto **Class** è come qualunque altro oggetto: pertanto potete ottenere un riferimento da esso e manipolarlo, esattamente come fa il caricatore. Uno dei modi per ottenere un riferimento a **Class** è il metodo **static forName()**, che accetta una **String** con il nome della classe di cui desiderate ottenere il riferimento: prestate estrema attenzione all'ortografia, nonché alle maiuscole e minuscole. Il riferimento alla classe restituito da **forName()** in questo caso viene ignorato, in quanto la chiamata viene effettuata per il suo effetto collaterale, ovvero il caricamento della classe **Gum** (qualora non sia già stata caricata). Nel corso del caricamento viene eseguita l'istruzione **static** di **Gum**.

Nell'esempio precedente, se il metodo **Class.forName()** non venisse eseguito con successo perché non riesce a trovare la classe da caricare, solleverebbe una **ClassNotFoundException**; in questo caso specifico, ci si è limitati a segnalare il problema e a continuare: in programmi più specializzati, potreste provare a risolvere il problema all'interno del gestore delle eccezioni.

Se volete utilizzare le informazioni di tipo al momento dell'esecuzione, dovrete in primo luogo ottenere un riferimento all'oggetto **Class** opportuno. **Class.forName()** è un sistema pratico per farlo, poiché non richiede l'effettiva presenza di un oggetto di quel tipo per ottenere il riferimento a **Class**. Tuttavia, se già disponete di un oggetto del tipo che vi interessa, potete procurarvi il riferimento a **Class** chiamando un metodo che fa parte della radice

di **Object**, **getClass()**. Questo metodo restituisce il riferimento a **Class** che rappresenta il tipo effettivo dell'oggetto. **Class** dispone di molti metodi interessanti, alcuni dei quali sono utilizzati di seguito.

```
//: typeinfo/toys/ToyTest.java
// Test della classe Class.
package typeinfo.toys;
import static net.mindview.util.Print.*;

interface HasBatteries {}
interface Waterproof {}
interface Shoots {}

class Toy {
  // Commentare il seguente costruttore predefinito
  // per visualizzare NoSuchMethodError da (*1*)
  Toy() {}
  Toy(int i) {}
}

class FancyToy extends Toy
implements HasBatteries, Waterproof, Shoots {
  FancyToy() { super(1); }
}

public class ToyTest {
  static void printInfo(Class cc) {
    print("Class name: " + cc.getName() +
      " is interface? [" + cc.isInterface() + "]");
    print("Simple name: " + cc.getSimpleName());
    print("Canonical name: " + cc.getCanonicalName());
  }
  public static void main(String[] args) {
    Class c = null;
    try {
      c = Class.forName("typeinfo.toys.FancyToy");
    } catch(ClassNotFoundException e) {
      print("Can't find FancyToy");
```

```
      System.exit(1);
    }
    printInfo(c);
    for(Class face : c.getInterfaces())
      printInfo(face);
    Class up = c.getSuperclass();
    Object obj = null;
    try {
      // Richiede il costruttore predefinito:
      obj = up.newInstance();
    } catch(InstantiationException e) {
      print("Cannot instantiate");
      System.exit(1);
    } catch(IllegalAccessException e) {
      print("Cannot access");
      System.exit(1);
    }
    printInfo(obj.getClass());
  }
} /* Output:
Class name: typeinfo.toys.FancyToy is interface? [false]
Simple name: FancyToy
Canonical name : typeinfo.toys.FancyToy
Class name: typeinfo.toys.HasBatteries is interface? [true]
Simple name: HasBatteries
Canonical name : typeinfo.toys.HasBatteries
Class name: typeinfo.toys.Waterproof is interface? [true]
Simple name: Waterproof
Canonical name : typeinfo.toys.Waterproof
Class name: typeinfo.toys.Shoots is interface? [true]
Simple name: Shoots
Canonical name : typeinfo.toys.Shoots
Class name: typeinfo.toys.Toy is interface? [false]
Simple name: Toy
Canonical name : typeinfo.toys.Toy
*///:~
```

**FancyToy** eredita da **Toy** e implementa (**implements**) le interfacce (**interface**) **HasBatteries**, **Waterproof** e **Shoots**. In **main()** viene creato un riferimento a **Class**, poi inizializzato alla classe **FancyToy** utilizzando **forName()** all'interno di un apposito blocco **try**. Ricordate che nella stringa passata a **forName()** è necessario utilizzare il nome esteso (*fully qualified name*), vale a dire completo di nome del pacchetto.

Il metodo **printInfo()** si serve di **getName()** per produrre il nome esteso della classe, e rispettivamente di **getSimpleName()** e **getCanonicalName()** (introdotti in Java SE5) per produrre il nome senza i riferimenti al pacchetto e, di nuovo, il nome esteso. Il metodo **isInterface()**, infine, segnala se l'oggetto **Class** corrente è un'interfaccia. Come vedete, mediante l'oggetto **Class** potete ottenere tutte le informazioni che vi occorrono su un tipo.

Il metodo **Class.getInterfaces()** chiamato in **main()** restituisce un array degli oggetti **Class** che rappresentano le interfacce presenti in un determinato oggetto **Class**.

Se avete un oggetto **Class**, potete anche conoscere il nome della sua classe di base diretta utilizzando **getSuperclass()**, in modo da recuperare il riferimento a una **Class** da sottoporre a ulteriore interrogazione: così facendo potete scoprire l'intera gerarchia di un oggetto in fase di esecuzione.

Il metodo **newInstance()** di **Class** è un modo per implementare un "costruttore virtuale", che vi consente di dire: "non so esattamente di che tipo sia l'oggetto, ma voglio che si crei automaticamente nel modo corretto". Nell'esempio precedente, **up** non è altro che un riferimento **Class** senza nessun'altra informazione di tipo nota al momento dell'esecuzione. Quando generate una nuova istanza viene restituito un riferimento **Object**, che tuttavia punta a un oggetto **Toy**. Naturalmente, prima che possiate trasmettere qualsiasi messaggio diverso da quelli accettati da **Object**, dovrete analizzare l'oggetto ed eseguire qualche operazione di conversione; inoltre, la classe che viene generata con **newInstance()** deve avere un costruttore predefinito. Nel prosieguo del capitolo vedrete come creare dinamicamente gli oggetti delle classi servendovi di qualsiasi costruttore, grazie alle API di riflessione Java.

**Esercizio 1** (1) In **ToyTest.java** commentate il costruttore predefinito di **Toy** e spiegate che cosa accade.

**Esercizio 2** (2) Incorporate un nuovo tipo di interfaccia in **ToyTest.java** e verificate che venga rilevato e visualizzato correttamente.

**Esercizio 3** (2) Aggiungete **Rhomboid** a **Shapes.java**. Create un **Rhomboid**, eseguitene l'upcast a **Shape**, quindi nuovamente il downcast a un **Rhomboid**. Infine provate a eseguire il downcast a un **Circle** e osservate che cosa accade.

**Esercizio 4** (2) Modificate l'esercizio precedente in modo che utilizzi **instanceof** per controllare il tipo prima di eseguire il downcasting.

**Esercizio 5** (3) Implementate un metodo **rotate(Shape)** in **Shapes.java**, che controlli per verificare se sta per eseguire la rotazione di un **Circle** e, in caso affermativo, non proceda all'operazione.

**Esercizio 6** (4) Modificate **Shapes.java** in modo che possa "evidenziare" (impostare un flag) tutte le forme di un determinato tipo. Il metodo **toString()** per ogni **Shape** derivata dovrà indicare se quella particolare **Shape** è "evidenziata".

**Esercizio 7** (3) Modificate **SweetShop.java** in modo che la creazione di qualsiasi tipo di oggetto sia controllata da un argomento da riga di comando: in pratica, se digitando **java SweetShop Candy** verrà generato soltanto l'oggetto **Candy**. Esaminate come è possibile controllare quali oggetti Class vengono caricati da riga di comando.

**Esercizio 8** (5) Scrivete un metodo che accetti un oggetto e visualizzi ricorsivamente tutte le classi nella gerarchia dell'oggetto.

**Esercizio 9** (5) Modificate l'esercizio precedente in modo che utilizzi **Class.getDeclaredFields()** per visualizzare anche le informazioni sui campi di una classe.

**Esercizio 10** (3) Scrivete un programma per determinare se un array di **Char** è un tipo primitivo o un **Object** effettivo.

## *Letterali di classe*

Java offre un altro modo per produrre il riferimento all'oggetto **Class**: il *letterale di classe* o *class literal*. Nel programma precedente sarebbe stata l'istruzione

```
FancyToy.class;
```

che non soltanto è più semplice ma anche più sicura, poiché viene controllata al momento della compilazione e pertanto non deve essere inserita in un blocco **try**; oltre a ciò, giacché elimina la necessità di una chiamata al metodo **forName()**, è anche più efficiente.

Il letterale di classe si applica sia alle normali classi sia alle interfacce, gli array e i tipi primitivi; inoltre ogni classe wrapper di primitivi possiede un campo standard denominato **TYPE**, che produce un riferimento all'oggetto **Class** per il tipo primitivo associato, tale che:

| ...sia equivalente a... | |
|---|---|
| **boolean.class** | **Boolean.TYPE** |
| **char.class** | **Character.TYPE** |
| **byte.class** | **Byte.TYPE** |
| **short.class** | **Short.TYPE** |
| **int.class** | **Integer.TYPE** |
| **long.class** | **Long.TYPE** |
| **float.class** | **Float.TYPE** |
| **double.class** | **Double.TYPE** |
| **void.class** | **Void.TYPE** |

L'autore raccomanda di utilizzare preferibilmente le versioni "**.class**", perché più coerenti con le normali classi.

È importante osservare che generando un riferimento a un oggetto **Class** con "**.class**" non ottenete automaticamente l'inizializzazione dell'oggetto. La preparazione di una classe all'utilizzo si compone infatti di tre fasi.

1. Caricamento (*Loading*), eseguito dal caricatore di classe, che trova i bytecode (di norma, ma non sempre, nel CLASSPATH del vostro disco) e crea un oggetto **Class** da essi.
2. Collegamento (*Linking*). La fase di "collegamento" verifica i bytecode nella classe, assegna lo spazio di registrazione per i campi **static** e, se necessario, risolve tutti i riferimenti ad altre classi trovati nella classe corrente.
3. Inizializzazione (*Initialization*). Se esiste una superclasse, viene inizializzata; vengono anche eseguiti gli inizializzatori **static** e i blocchi **static** di inizializzazione.

L'inizializzazione è posticipata fino al primo riferimento a un metodo **static** o a un campo **static** non costante; ricordate che il costruttore è implicitamente **static**.

```
//: typeinfo/ClassInitialization.java
import java.util.*;

class Initable {
  static final int staticFinal = 47;
  static final int staticFinal2 =
    ClassInitialization.rand.nextInt(1000);
```

```
  static {
    System.out.println("Initializing Initable");
  }
}

class Initable2 {
  static int staticNonFinal = 147;
  static {
    System.out.println("Initializing Initable2");
  }
}

class Initable3 {
  static int staticNonFinal = 74;
  static {
    System.out.println("Initializing Initable3");
  }
}

public class ClassInitialization {
  public static Random rand = new Random(47);
  public static void main(String[] args) throws Exception {
    Class initable = Initable.class;
    System.out.println("After creating Initable ref");
    // Non attiva l'inizializzazione:
    System.out.println(Initable.staticFinal);
    // Attiva l'inizializzazione:
    System.out.println(Initable.staticFinal2);
    // Attiva l'inizializzazione:
    System.out.println(Initable2.staticNonFinal);
    Class initable3 = Class.forName("Initable3");
    System.out.println("After creating Initable3 ref");
    System.out.println(Initable3.staticNonFinal);
  }
} /* Output:
After creating Initable ref
47
```

```
Initializing Initable
258
Initializing Initable2
147
Initializing Initable3
After creating Initable3 ref
74
*///:~
```

Per ragioni legate all'efficienza, l'inizializzazione è il più possibile "pigra". Dalla creazione del riferimento **initable** potete vedere che il semplice utilizzo della sintassi **.class** per ottenere un riferimento alla classe non provoca l'inizializzazione. Tuttavia, **Class.forName()** inizializza immediatamente la classe per produrre il riferimento a **Class**, come mostrato nella creazione di **initable3**.

Se un valore **static final** è una "costante di compilazione" come **Initable.staticFinal**, potrà essere letto senza provocare l'inizializzazione della classe **Initable**. Tuttavia, il fatto di rendere un campo **static** e **final** non garantisce questo comportamento: l'accesso a **Initable.staticFinal2** provoca l'inizializzazione della classe, poiché **Initable.staticFinal2** non può essere una costante.

Se un campo **static** non è **final**, il suo accesso richiede comunque il collegamento (per assegnare lo spazio di registrazione per il campo) e l'inizializzazione (per inizializzare questo spazio) prima della lettura, come è evidente nel codice di accesso a **Initable2.staticNonFinal**.

## *Riferimenti generici di classi*

Un riferimento a **Class** punta a un oggetto **Class**, che produce le istanze delle classi e contiene tutto il codice di metodo per queste istanze, nonché i membri **static** della classe in questione. Quindi, in realtà, un riferimento **Class** indica il tipo esatto di ciò a cui fa riferimento: un oggetto della classe **Class**.

Tuttavia i progettisti di Java SE5 hanno approfittato dell'occasione per rendere questo meccanismo un poco più specifico, permettendovi di limitare il tipo di oggetto **Class** al quale punta il riferimento **Class**, per mezzo di una sintassi generica. Nel seguente esempio, entrambe le sintassi sono corrette.

```
//: typeinfo/GenericClassReferences.java

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

```
    Class intClass = int.class;
    Class<Integer> genericIntClass = int.class;
    genericIntClass = Integer.class; // Stessa operazione
    intClass = double.class;
    // genericIntClass = double.class; // Illegale
  }
} ///:~
```

Il normale riferimento di classe non produce un avvertimento da parte del compilatore. Tuttavia, potete notare che il normale riferimento può essere riassegnato a qualunque altro oggetto **Class**, mentre il riferimento generico di classe può essere assegnato soltanto al suo tipo dichiarato. Utilizzando la sintassi generica, permettete che il compilatore implementi questo controllo aggiuntivo sui tipi.

E se voleste allentare in qualche modo questo vincolo? A prima vista sembra che sia possibile scrivere:

```
Class<Number> genericNumberClass = int.class;
```

Ciò sembrerebbe avere senso poiché **Integer** è ereditato da **Number**, tuttavia l'istruzione non funzionerà perché l'oggetto **Integer Class** non è una sottoclasse dell'oggetto **Number Class**: nel Capitolo 3 avrete modo di esaminare a fondo questa distinzione apparentemente sottile.

Per allentare i vincoli quando si utilizzano i riferimenti **Class** generici, l'autore ricorre ai metacaratteri (*wildcard*), che sono parte integrante dei generici di Java: il metacarattere utilizzato è "**?**", ed equivale a "qualsiasi cosa". Pertanto potete aggiungere il metacarattere al normale riferimento **Class** dell'esempio precedente e ottenere gli stessi risultati.

```
//: typeinfo/WildcardClassReferences.java

public class WildcardClassReferences {
  public static void main(String[] args) {
    Class<?> intClass = int.class;
    intClass = double.class;
  }
} ///:~
```

In Java SE5, la forma **Class<?>** è preferibile rispetto al **Class** normale, benché siano equivalenti e il **Class** normale, come avete visto, non generi avvertimenti da parte del compilatore. Il vantaggio di utilizzare **Class<?>** è che indica che non state utilizzando un riferimento di classe non specifico per caso o per ignoranza della sintassi, ma che avete scelto espressamente la variante non specifica.

Allo scopo di generare un riferimento **Class** limitato a un tipo o a qualsiasi sottotipo, combinate il metacarattere **?** con la parola chiave **extend** per generare un limite; in pratica, anziché **Class<Number>**, scriverete:

```
//: typeinfo/BoundedClassReferences.java

public class BoundedClassReferences {
  public static void main(String[] args) {
    Class<? extends Number> bounded = int.class;
    bounded = double.class;
    bounded = Number.class;
    // O qualunque altra cosa derivata da Number.
  }
} ///:~
```

Il motivo dell'aggiunta della sintassi generica ai riferimenti **Class** è soltanto quello di garantire il controllo di tipo al momento della compilazione, in modo che in caso di errore questo venga evidenziato il più rapidamente possibile.

Ecco un esempio che utilizza la sintassi di classe generica, registrando un riferimento di classe e successivamente elaborando una **List** popolata di oggetti generati con **newInstance()**.

```
//: typeinfo/FilledList.java
import java.util.*;

class CountedInteger {
  private static long counter;
  private final long id = counter++;
  public String toString() { return Long.toString(id); }
}
```

```
public class FilledList<T> {
  private Class<T> type;
  public FilledList(Class<T> type) { this.type = type; }
  public List<T> create(int nElements) {
    List<T> result = new ArrayList<T>();
    try {
      for(int i = 0; i < nElements; i++)
        result.add(type.newInstance());
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
    return result;
  }
  public static void main(String[] args) {
    FilledList<CountedInteger> fl =
      new FilledList<CountedInteger>(CountedInteger.class);
    System.out.println(fl.create(15));
  }
} /* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
*///:~
```

Notate che questa classe presuppone che qualunque tipo essa gestisca abbia un costruttore predefinito (privo di argomenti) e che in caso contrario venga sollevata un'eccezione: di per sé, il compilatore non produce alcun avviso per questo programma.

Un effetto interessante derivante dall'utilizzo della sintassi generica per gli oggetti **Class** è che **newInstance()** restituisce il tipo di oggetto esatto anziché un **Object** di base, come avete visto in **ToyTest.java**. Questo comportamento è piuttosto limitativo.

```
//: typeinfo/toys/GenericToyTest.java
// Test della class Class.
package typeinfo.toys;
public class GenericToyTest {
  public static void main(String[] args) throws Exception {
    Class<FancyToy> ftClass = FancyToy.class;
```

```
    // Produce il tipo esatto:
    FancyToy fancyToy = ftClass.newInstance();
    Class<? super FancyToy> up = ftClass.getSuperclass();
    // Non compilera':
    // Class<Toy> up2 = ftClass.getSuperclass();
    // Produce soltanto Object:
    Object obj = up.newInstance();
  }
} ///:~
```

Se ottenete la superclasse, il compilatore vi permetterà di affermare soltanto che il riferimento a essa è "una qualche classe che è una superclasse di **FancyToy**", come mostra l'espressione **Class<? super FancyToy>**. Non accetterà una dichiarazione **Class<Toy>**, il che sembra strano poiché **getSuperclass()** restituisce la classe di base (non l'interfaccia) e il compilatore sa perfettamente di che cosa si tratta, al momento della compilazione: quindi, in questo caso sa che si tratta di **Toy.class** e non semplicemente di "una qualche superclasse di **FancyToy**". In ogni caso, a causa di questa indeterminazione il valore restituito da **up.newInstance()** non è un tipo preciso, ma soltanto un **Object**.

### *Nuova sintassi di cast*

Java SE5 aggiunge anche una sintassi di casting da utilizzare con i riferimenti **Class**, ossia il metodo **cast()**.

```
//: typeinfo/ClassCasts.java

class Building {}
class House extends Building {}

public class ClassCasts {
  public static void main(String[] args) {
    Building b = new House();
    Class<House> houseType = House.class;
    House h = houseType.cast(b);
    h = (House)b; // ... oppure fate semplicemente questo.
  }
} ///:~
```

Il metodo **cast()** converte al tipo del riferimento di **Class** l'oggetto passato come argomento. Ovviamente se osservate il codice dell'esempio precedente vi sembrerà di avere scritto molto più codice, in confronto all'ultima riga del metodo **main()**, che esegue la stessa operazione.

La nuova sintassi di casting è utile nelle situazioni in cui non è possibile utilizzare un normale cast, di solito quando scrivete codice generico (si veda il Capitolo 3) e avete archiviato un riferimento **Class** che volete utilizzare in seguito per il casting.

È una situazione piuttosto rara: nell'intera libreria di Java SE5 l'autore ha identificato una sola situazione in cui è stato utilizzato il metodo **cast()**, precisamente in **com.sun.mirror.util.DeclarationFilter**.

Un'altra nuova caratteristica di Java SE5 che ancora non è stata utilizzata nella libreria di Java SE 5 è il metodo **Class.asSubclass()**. Esso consente di eseguire il casting dell'oggetto di classe a un tipo più specifico.

## Controlli pre-casting

Finora, avete visto varie forme di RTTI, tra cui:

1. il casting classico; per esempio, **"(Shape)"** che utilizza l'RTTI per assicurarsi che il casting sia corretto; può sollevare un'eccezione **ClassCastException** se tentate di eseguire un casting errato;
2. l'oggetto **Class** che rappresenta il tipo del vostro oggetto; l'oggetto **Class** può essere utilizzato per ottenere informazioni di runtime utili.

In C++ il casting classico **"(Shape)"** non esegue operazioni RTTI: semplicemente indica al compilatore di considerare l'oggetto come il nuovo tipo. In Java, che esegue il controllo del tipo, questa operazione viene identificata come "casting di tipo sicuro" (*type-safe casting*). Come ricorderete, il motivo del termine *downcast* risiede nella disposizione classica del diagramma di ereditarietà gerarchico: se il casting di un **Circle** a una **Shape** è un upcast, quello da una **Shape** a un **Circle** è un downcast.

Tuttavia il compilatore, sapendo che un **Circle** è anche una **Shape**, permette l'assegnazione di upcasting senza richiedere alcuna sintassi di casting esplicita. Il compilatore non può sapere, data una **Shape**, che cosa sia quella forma in realtà: potrebbe essere esattamente una **Shape** o un sottotipo di **Shape**, come **Circle**, **Square**, **Triangle** o qualche altro. In fase di compilazione il compilatore vede soltanto una **Shape**, pertanto non consentirà di eseguire un'assegnazione di downcast senza utilizzare un casting esplicito: mediante questa tecnica, in-

vece, il compilatore sa che disponete di informazioni supplementari che vi consentono di sapere che il tipo è particolare. Il compilatore eseguirà un controllo per verificare se il downcast è possibile, così da non permettervi di eseguire il downcast a un tipo che non sia effettivamente una sottoclasse.

Esiste una terza forma di RTTI in Java: si tratta della parola chiave **instanceof**, che indica se un oggetto è un'istanza di un particolare tipo. Restituisce un valore **boolean** in modo da poter essere utilizzato sotto forma di interrogazione, come nell'esempio seguente.

```
if(x instanceof Dog)
  ((Dog)x).bark();
```

L'istruzione **if** verifica se l'oggetto **x** appartiene alla classe **Dog** prima di convertire **x** in un **Dog**. È importante utilizzare **instanceof** prima di un downcast, quando non disponete di altre informazioni che indichino il tipo dell'oggetto: in caso contrario verrà prodotta una **ClassCastException**.

Come spesso accade, potreste essere alla ricerca di un tipo, per esempio dei triangoli da evidenziare: in tal caso, utilizzando **instanceof** potete facilmente contrassegnare tutti gli oggetti che vi occorrono. Immaginate una famiglia di classi che descrivono animali da compagnia (**Pet**) e i loro proprietari, una caratteristica che risulterà utile in un esempio successivo. Ogni **Individual** nella gerarchia possiede un identificativo **id** e un nome opzionale. Sebbene le classi seguenti ereditino da **Individual**, questa classe presenta alcune complicazioni, pertanto il codice verrà illustrato e analizzato nel Capitolo 5. Come potete vedere, a questo punto non è effettivamente necessario conoscere il codice di **Individual**: vi basta soltanto sapere che potete creare l'oggetto con o senza un nome, e che ogni **Individual** ha un metodo **id()**, che restituisce un identificativo univoco creato contando ciascun oggetto. Esiste anche un metodo **toString()**: se non fornite un nome per un oggetto **Individual**, **toString()** ne genererà una versione semplice.

Di seguito è mostrata la gerarchia di classi che eredita da **Individual**.

```
//: typeinfo/pets/Person.java
package typeinfo.pets;

public class Person extends Individual {
  public Person(String name) { super(name); }
} ///:~

//: typeinfo/pets/Pet.java
```

```
package typeinfo.pets;

public class Pet extends Individual {
  public Pet(String name) { super(name); }
  public Pet() { super(); }
} ///:~

//: typeinfo/pets/Dog.java
package typeinfo.pets;

public class Dog extends Pet {
  public Dog(String name) { super(name); }
public Dog() { super(); }
} ///:~

//: typeinfo/pets/Mutt.java
package typeinfo.pets;

public class Mutt extends Dog {
  public Mutt(String name) { super(name); }
  public Mutt() { super(); }
} ///:~

//: typeinfo/pets/Pug.java
package typeinfo.pets;

public class Pug extends Dog {
  public Pug(String name) { super(name); }
  public Pug() { super(); }
} ///:~

//: typeinfo/pets/Cat.java
package typeinfo.pets;

public class Cat extends Pet {
  public Cat(String name) { super(name); }
  public Cat() { super(); }
} ///:~

//: typeinfo/pets/EgyptianMau.java
```

```
package typeinfo.pets;

public class EgyptianMau extends Cat {
  public EgyptianMau(String name) { super(name); }
  public EgyptianMau() { super(); }
} ///:~

//: typeinfo/pets/Manx.java
package typeinfo.pets;

public class Manx extends Cat {
  public Manx(String name) { super(name); }
  public Manx() { super(); }
} ///:~

//: typeinfo/pets/Cymric.java
package typeinfo.pets;

public class Cymric extends Manx {
  public Cymric(String name) { super(name); }
  public Cymric() { super(); }
} ///:~

//: typeinfo/pets/Rodent.java
package typeinfo.pets;

public class Rodent extends Pet {
  public Rodent(String name) { super(name); }
  public Rodent() { super(); }
} ///:~

//: typeinfo/pets/Rat.java
package typeinfo.pets;

public class Rat extends Rodent {
  public Rat(String name) { super(name); }
  public Rat() { super(); }
} ///:~

//: typeinfo/pets/Mouse.java
```

```
package typeinfo.pets;

public class Mouse extends Rodent {
  public Mouse(String name) { super(name); }
  public Mouse() { super(); }
} ///:~

//: typeinfo/pets/Hamster.java
package typeinfo.pets;

public class Hamster extends Rodent {
  public Hamster(String name) { super(name); }
  public Hamster() { super(); }
} ///:~
```

È poi necessario un meccanismo al fine di generare casualmente vari tipi di animali e, per praticità, per creare array e **List** di **Pet**; l'evoluzione di questo strumento con le varie implementazioni sarà garantita definendolo come classe astratta.

```
//: typeinfo/pets/PetCreator.java
// Crea sequenze casuali di Pet.
package typeinfo.pets;
import java.util.*;
public abstract class PetCreator {
private Random rand = new Random(47);
  // La List dei diversi tipi di Pet da creare:
  public abstract List<Class<? extends Pet>> types();
  public Pet randomPet() { // Crea un Pet casuale
    int n = rand.nextInt(types().size());
    try {
      return types().get(n).newInstance();
    } catch(InstantiationException e) {
      throw new RuntimeException(e);
    } catch(IllegalAccessException e) {
      throw new RuntimeException(e);
    }
  }
```

```
  public Pet[] createArray(int size) {
    Pet[] result = new Pet[size];
    for(int i = 0; i < size; i++)
      result[i] = randomPet();
    return result;
  }
  public ArrayList<Pet> arrayList(int size) {
    ArrayList<Pet> result = new ArrayList<Pet>();
    Collections.addAll(result, createArray(size));
    return result;
  }
} ///:~
```

Il metodo **abstract types()** si affida a una classe derivata per ottenere la **List** di oggetti **Class**: questa è una variante del design pattern *Template Method*. Notate che il tipo di classe è indicato come "qualsiasi cosa derivata da Pet", in modo che **newInstance()** produca un **Pet** senza richiedere un cast.

Il metodo **randomPet()** genera casualmente indici nella **List** e utilizza l'oggetto **Class** selezionato per generare una nuova istanza di quella classe, con **Class.newInstance()**. Il metodo **createArray()** utilizza **randomPet()** per popolare un array e, a sua volta, **arrayList()** utilizza **createArray()**.

In seguito alla chiamata a **newInstance()** potete ottenere due tipi di eccezione, che sono gestite nelle clausole **catch** successive al blocco **try**. Come sempre, i nomi delle eccezioni sono descrittivi, utili per capire che cosa non va: **IllegalAccessException**, per esempio, fa riferimento a una violazione del meccanismo di sicurezza di Java, in questo caso il fatto che il costruttore predefinito sia **private**.

Quando derivate una sottoclasse di **PetCreator**, l'unica cosa che dovete fornire è la **List** dei tipi di **Pet** da generare utilizzando **randomPet()** e gli altri metodi: il metodo **types()** restituirà un riferimento a una **List** statica. Di seguito è mostrata un'implementazione che si serve di **forName**.

```
//: typeinfo/pets/ForNameCreator.java
package typeinfo.pets;
import java.util.*;

public class ForNameCreator extends PetCreator {
```

```
  private static List<Class<? extends Pet>> types =
    new ArrayList<Class<? extends Pet>>();
  // I tipi da creare casualmente:
  private static String[] typeNames = {
    "typeinfo.pets.Mutt",
    "typeinfo.pets.Pug",
    "typeinfo.pets.EgyptianMau",
    "typeinfo.pets.Manx",
    "typeinfo.pets.Cymric",
    "typeinfo.pets.Rat",
    "typeinfo.pets.Mouse",
    "typeinfo.pets.Hamster"
  };
  @SuppressWarnings("unchecked")
  private static void loader() {
    try {
      for(String name : typeNames)
        types.add(
          (Class<? extends Pet>)Class.forName(name));
    } catch(ClassNotFoundException e) {
      throw new RuntimeException(e);
    }
  }
  static { loader(); }
  public List<Class<? extends Pet>> types() {return types;}
} ///:~
```

Il metodo **loader()** crea la **List** di oggetti **Class** utilizzando **Class.forName()**; quest'ultimo metodo può generare una **ClassNotFoundException**, dovuta al passaggio di una **String** che potrebbe non essere convalidata in fase di esecuzione. Poiché gli oggetti **Pet** sono nel package **typeinfo**, per fare riferimento alle classi deve essere utilizzato il nome del package.

Per produrre una **List** tipizzata di oggetti **Class** è necessario un cast, che genera un messaggio di warning in fase di compilazione. Il metodo **loader()** è definito separatamente e successivamente inserito in una clausola di inizializzazione statica, poiché l'annotazione **@SuppressWarnings** non può essere inserita direttamente in una clausola di inizializzazione statica.

Per contare gli oggetti **Pet** è necessario uno strumento che tenga traccia delle quantità dei diversi tipi di animali generati. Un oggetto **Map** è perfetto per questo scopo: le chiavi sono i nomi del tipo di **Pet** e i valori sono **Integer**, in cui vengono registrate le quantità di **Pet**; questo consente di determinare, per esempio, quanti oggetti **Hamster** sono stati creati. Per contare i **Pet** si può utilizzare **instanceof**.

```
//: typeinfo/PetCount.java
// Utilizzo di instanceof.
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class PetCount {
  static class PetCounter extends HashMap<String,Integer> {
    public void count(String type) {
      Integer quantity = get(type);
      if(quantity == null)
        put(type, 1);
      else
        put(type, quantity + 1);
    }
  }
  public static void
  countPets(PetCreator creator) {
    PetCounter counter= new PetCounter();
    for(Pet pet : creator.createArray(20)) {
      // Elenca ogni pet:
      printnb(pet.getClass().getSimpleName() + " ");
      if(pet instanceof Pet)
        counter.count("Pet");
      if(pet instanceof Dog)
        counter.count("Dog");
      if(pet instanceof Mutt)
        counter.count("Mutt");
      if(pet instanceof Pug)
        counter.count("Pug");
```

```
      if(pet instanceof Cat)
        counter.count("Cat");
      if(pet instanceof Manx)
        counter.count("EgyptianMau");
      if(pet instanceof Manx)
        counter.count("Manx");
      if(pet instanceof Manx)
        counter.count("Cymric");
      if(pet instanceof Rodent)
        counter.count("Rodent");
      if(pet instanceof Rat)
        counter.count("Rat");
      if(pet instanceof Mouse)
        counter.count("Mouse");
      if(pet instanceof Hamster)
        counter.count("Hamster");
    }
    // Visualizza il conteggio:
    print();
    print(counter);
  }
  public static void main(String[] args) {
    countPets(new ForNameCreator());
  }
} /* Output:
Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat
EgyptianMau Hamster EgyptianMau Mutt Mutt Cymric Mouse Pug
Mouse Cymric
{Pug=3, Cat=9, Hamster=1, Cymric=7, Mouse=2, Mutt=3,
Rodent=5, Pet=20, Manx=7, EgyptianMau=7, Dog=6, Rat=2}
*///:~
```

In **countPets()** un array viene popolato casualmente con **Pet** tramite **PetCreator**, quindi ogni **Pet** nell'array viene esaminato e contato utilizzando **instanceof**.

L'operatore **instanceof** è soggetto a un vincolo piuttosto restrittivo, dal momento che consente di eseguire confronti solo con i tipi, non con gli oggetti

**Class**. Nell'esempio precedente potreste ritenere noioso scrivere tante espressioni **instanceof** e, in effetti, avete ragione. Tuttavia non esiste alcun modo per automatizzare in modo brillante **instanceof** creando un array di oggetti **Class** e confrontandolo con questi: in ogni caso, tra poco vedrete un'alternativa. Questo vincolo non è un limite così negativo come si potrebbe pensare poiché, alla fine, se vi trovate a scrivere numerose espressioni **instanceof** la struttura del vostro progetto avrà probabilmente alcuni difetti.

## *Utilizzo dei letterali di classe*

Se reimplementate **PetCreator** utilizzando il letterale di classe, il risultato risulterà più chiaro.

```
//: typeinfo/pets/LiteralPetCreator.java
// Utilizzo dei letterali di classe.
package typeinfo.pets;
import java.util.*;

public class LiteralPetCreator extends PetCreator {
  // Non e' necessario nessun blocco try.
  @SuppressWarnings("unchecked")
  public static final List<Class<? extends Pet>> allTypes =
    Collections.unmodifiableList(Arrays.asList(
      Pet.class, Dog.class, Cat.class,  Rodent.class,
      Mutt.class, Pug.class, EgyptianMau.class, Manx.class,
      Cymric.class, Rat.class, Mouse.class,Hamster.class));
  // Tipi per la creazione casuale:
  private static final List<Class<? extends Pet>> types =
    allTypes.subList(allTypes.indexOf(Mutt.class),
      allTypes.size());
  public List<Class<? extends Pet>> types() {
    return types;
  }
  public static void main(String[] args) {
    System.out.println(types);
  }
} /* Output:
[class typeinfo.pets.Mutt, class typeinfo.pets.Pug, class
```

```
typeinfo.pets.EgyptianMau, class typeinfo.pets.Manx, class
typeinfo.pets.Cymric, class typeinfo.pets.Rat, class
typeinfo.pets.Mouse, class typeinfo.pets.Hamster]
*///:~
```

Nell'esempio **PetCount3.java**, che vedrete tra breve, è necessario precaricare una **Map** con tutti i tipi **Pet**, non soltanto quelli che devono essere generati casualmente; quindi è necessario specificare **allTypes List**. L'elenco **types** è la porzione di **allTypes**, generata con **List.subList()**, che include i tipi esatti di animali, pertanto viene utilizzata per la generazione casuale di **Pet**.

Questa volta la creazione di **types** non deve essere controllata da un blocco **try** poiché è valutata in fase di compilazione e, a differenza di **Class.forName()**, così non solleva alcuna eccezione.

Ora disponete di due implementazioni di **PetCreator** nella libreria **typeinfo.pets**. Per fare in modo che la seconda sia l'implementazione predefinita potete generare una cosiddetta *façade*, ovvero un'applicazione "di facciata" che utilizza **LiteralPetCreator**.

```
//: typeinfo/pets/Pets.java
// Facade per produrre un PetCreator predefinito.
package typeinfo.pets;
import java.util.*;

public class Pets {
  public static final PetCreator creator =
    new LiteralPetCreator();
  public static Pet randomPet() {
    return creator.randomPet();
  }
  public static Pet[] createArray(int size) {
    return creator.createArray(size);
  }
  public static ArrayList<Pet> arrayList(int size) {
    return creator.arrayList(size);
  }
} ///:~
```

Questa fornisce anche l'indirezione per **randomPet()**, **createArray()** e **array List()**.

Poiché **PetCount.countPets()** accetta un argomento di tipo **PetCreator**, è possibile testare facilmente **LiteralPetCreator**, ricorrendo alla suddetta facciata:

```
//: typeinfo/PetCount2.java
import typeinfo.pets.*;

public class PetCount2 {
  public static void main(String[] args) {
    PetCount.countPets(Pets.creator);
  }
} /* (Da eseguire per visualizzare l'output) *///:~
```

L'output è identico a quello di **PetCount.java**.

## *Operatore instanceof dinamico*

Il metodo **Class.isInstance**() fornisce un meccanismo per verificare dinamicamente il tipo di oggetto, permettendo così di rimuovere tutte le tediose dichiarazioni **instanceof** da **PetCount.java**.

```
//: typeinfo/PetCount3.java
// Utilizzo di isInstance()
import typeinfo.pets.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class PetCount3 {
  static class PetCounter
  extends LinkedHashMap<Class<? extends Pet>,Integer> {
    public PetCounter() {
      super(MapData.map(LiteralPetCreator.allTypes, 0));
    }
    public void count(Pet pet) {
      // Class.isInstance() elimina la necessita' di
```

```
      // instanceof:
      for(Map.Entry<Class<? extends Pet>,Integer> pair
          : entrySet())
        if(pair.getKey().isInstance(pet))
          put(pair.getKey(), pair.getValue() + 1);
    }
    public String toString() {
      StringBuilder result = new StringBuilder("{");
      for(Map.Entry<Class<? extends Pet>,Integer> pair
          : entrySet()) {
        result.append(pair.getKey().getSimpleName());
        result.append("=");
        result.append(pair.getValue());
        result.append(", ");
      }
      result.delete(result.length()-2, result.length());
      result.append("}");
      return result.toString();
    }
  }
  public static void main(String[] args) {
    PetCounter petCount = new PetCounter();
    for(Pet pet : Pets.createArray(20)) {
      printnb(pet.getClass().getSimpleName() + " ");
      petCount.count(pet);
    }
    print();
    print(petCount);
  }
} /* Output:
Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat
EgyptianMau Hamster EgyptianMau Mutt Mutt Cymric Mouse Pug
Mouse Cymric
{Pet=20, Dog=6, Cat=9, Rodent=5, Mutt=3, Pug=3,
EgyptianMau=2, Manx=7, Cymric=5, Rat=2, Mouse=2, Hamster=1}
*///:~
```

Per contare tutti i diversi tipi di **Pet**, viene precaricata la mappa **PetCounter** con i tipi ottenuti da **LiteralPetCreator.allTypes**. Quest'ultima si serve della classe **net.mindview.util.MapData** che accetta un oggetto **Iterable** (**allTypes List**) e un valore costante (zero, in questo caso), e popola **Map** con le chiavi prese da **allTypes** e i valori zero. Senza precaricare la **Map** sareste limitati al conteggio dei tipi generati casualmente, escludendo quelli di base come **Pet** e **Cat.**

Potete notare che il metodo **isInstance()** ha eliminato l'esigenza delle espressioni **instanceof**. Inoltre questo significa che potete aggiungere nuovi tipi di **Pet** semplicemente cambiando l'array **LiteralPetCreator.types**; il resto del programma non dovrà essere modificato, come avverrebbe invece con **instanceof**.

Il metodo **toString()** è stato sovraccaricato per fornire un output di più facile lettura, che corrisponda a quello tipico visualizzato da una **Map**.

## *Conteggio ricorsivo*

La **Map** in **PetCount3.PetCounter** è stata precaricata con tutti i **Pet** delle classi. In alternativa al precaricamento della mappa, potete utilizzare **Class.isAssignableFrom()** e creare un'utility multiuso che non si limiti a contare i **Pet**.

```
//: net/mindview/util/TypeCounter.java
// Conta le istanze di una famiglia di tipi.
package net.mindview.util;
import java.util.*;

public class TypeCounter extends HashMap<Class<?>,Integer>{
  private Class<?> baseType;
  public TypeCounter(Class<?> baseType) {
    this.baseType = baseType;
  }
  public void count(Object obj) {
    Class<?> type = obj.getClass();
    if(!baseType.isAssignableFrom(type))
      throw new RuntimeException(obj + " incorrect type: "
        + type + ", should be type or subtype of "
        + baseType);
    countClass(type);
  }
```

```
  private void countClass(Class<?> type) {
    Integer quantity = get(type);
    put(type, quantity == null ? 1 : quantity + 1);
    Class<?> superClass = type.getSuperclass();
    if(superClass != null &&
       baseType.isAssignableFrom(superClass))
      countClass(superClass);
  }
  public String toString() {
    StringBuilder result = new StringBuilder("{");
    for(Map.Entry<Class<?>,Integer> pair : entrySet()) {
      result.append(pair.getKey().getSimpleName());
      result.append("=");
      result.append(pair.getValue());
      result.append(", ");
    }
    result.delete(result.length()-2, result.length());
    result.append("}");
    return result.toString();
  }
} ///:~
```

Il metodo **count()**, tramite **obj.getClass()**, ottiene la **Class** dell'oggetto passato come argomento e utilizza **isAssignableFrom()** per eseguire un controllo al momento dell'esecuzione, verificando che l'oggetto passato appartenga effettivamente alla gerarchia in questione. Per prima cosa, **countClass()** conta il tipo esatto della classe: in questo modo se **baseType** è assegnabile dalla superclasse, cioè se **baseType.isAssignableFrom(superClass)** restituisce **true**, il metodo **countClass()** viene chiamato ricorsivamente sulla superclasse.

```
//: typeinfo/PetCount4.java
import typeinfo.pets.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class PetCount4 {
  public static void main(String[] args) {
```

```
      TypeCounter counter = new TypeCounter(Pet.class);
      for(Pet pet : Pets.createArray(20)) {
        printnb(pet.getClass().getSimpleName() + " ");
        counter.count(pet);
      }
  print();
      print(counter);
    }
  } /* Output: (esempio)
  Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat
  EgyptianMau Hamster EgyptianMau Mutt Mutt Cymric Mouse Pug
  Mouse Cymric
  {Mouse=2, Dog=6, Manx=7, EgyptianMau=2, Rodent=5, Pug=3,
  Mutt=3, Cymric=5, Cat=9, Hamster=1, Pet=20, Rat=2}
  *///:~
```

Come potete vedere dall'output, vengono contati sia i tipi di base sia i tipi esatti.

**Esercizio 11** (2) Aggiungete un **Gerbil** alla libreria **typeinfo.pets** e modificate tutti gli esempi in questo capitolo per adattarli a questa nuova classe.

**Esercizio 12** (3) Utilizzate **TypeCounter** con la classe **CoffeeGenerator.java** del Capitolo 3.

**Esercizio 13** (3) Utilizzate **TypeCounter** con l'esempio **RegisteredFactories.java** di questo capitolo.

## Factory registrate

Un problema che si presenta con la generazione degli oggetti della gerarchia **Pet** è il fatto che ogni volta che vi aggiungete un nuovo tipo di **Pet** dovete ricordare di aggiungerlo anche alle voci in **LiteralPetCreator.java**. In un'applicazione in cui aggiungete regolarmente più classi questa operazione può risultare problematica.

Potreste pensare di aggiungere un'inizializzatore **static** a ogni sottoclasse, in modo che l'inizializzatore aggiunga la relativa classe a una lista prevista a questo scopo. Purtroppo gli inizializzatori statici vengono chiamati soltanto in occasione del primo caricamento della classe, pertanto vi ritroverete alle

prese con il classico dilemma dell'uovo e della gallina: non esiste la classe nell'elenco del generatore, che quindi non potrà mai creare un oggetto di quella classe, pertanto quest'ultima non potrà mai essere caricata e posta nell'elenco.

In pratica, siete costretti a creare la lista voi stessi scrivendo ogni volta il codice necessario, a meno che non vogliate realizzare uno strumento che cerchi e analizzi il vostro codice sorgente e quindi crei e compili la lista. Dunque la cosa migliore che potete fare è forse inserire la lista in un punto centrale, ben in evidenza. La classe di base per la gerarchia corrente è probabilmente il posto migliore.

L'altra modifica che eseguirete è rinviare la creazione dell'oggetto alla classe stessa, utilizzando il design pattern *Factory Method*. Un metodo di questo tipo può essere chiamato in modo polimorfico e crea automaticamente un oggetto del tipo adatto. In questa versione molto semplice, il Factory Method è il metodo **create()** nell'interfaccia **Factory**.

```
//: typeinfo/factory/Factory.java
package typeinfo.factory;
public interface Factory<T> { T create(); } ///:~
```

Il parametro generico **T** fa sì che **create()** restituisca un tipo differente per ogni implementazione di **Factory**. Questo metodo si serve anche di tipi di ritorno covarianti.

In questo esempio, la classe di base **Part** contiene una **List** di oggetti factory. Le factory per i tipi che dovrebbero essere prodotti con il metodo **createRandom()** vengono "registrate" tramite la classe di base, aggiungendole alla **List partFactories** e diventando le cosiddette "factory registrate" (*Registered Factory*).

```
//: typeinfo/RegisteredFactories.java
// Registrazione delle factory di classe nella classe di base.
import typeinfo.factory.*;
import java.util.*;

class Part {
  public String toString() {
    return getClass().getSimpleName();
  }
```

```
  static List<Factory<? extends Part>> partFactories =
    new ArrayList<Factory<? extends Part>>();
  static {
    // Collections.addAll() visualizza un avvertimento di tipo
    // "unchecked generic array creation ... for varargs
    // parameter".
    partFactories.add(new FuelFilter.Factory());
    partFactories.add(new AirFilter.Factory());
    partFactories.add(new CabinAirFilter.Factory());
    partFactories.add(new OilFilter.Factory());
    partFactories.add(new FanBelt.Factory());
    partFactories.add(new PowerSteeringBelt.Factory());
    partFactories.add(new GeneratorBelt.Factory());
  }
  private static Random rand = new Random(47);
  public static Part createRandom() {
    int n = rand.nextInt(partFactories.size());
    return partFactories.get(n).create();
  }
}

class Filter extends Part {}

class FuelFilter extends Filter {
  // Crea una factory di classe per ogni tipo specificato:
  public static class Factory
  implements typeinfo.factory.Factory<FuelFilter> {
    public FuelFilter create() { return new FuelFilter(); }
  }
}

class AirFilter extends Filter {
  public static class Factory
  implements typeinfo.factory.Factory<AirFilter> {
    public AirFilter create() { return new AirFilter(); }
  }
```

```
}

class CabinAirFilter extends Filter {
  public static class Factory
  implements typeinfo.factory.Factory<CabinAirFilter> {
    public CabinAirFilter create() {
      return new CabinAirFilter();
    }
  }
}

class OilFilter extends Filter {
  public static class Factory
  implements typeinfo.factory.Factory<OilFilter> {
    public OilFilter create() { return new OilFilter(); }
  }
}

class Belt extends Part {}

class FanBelt extends Belt {
  public static class Factory
  implements typeinfo.factory.Factory<FanBelt> {
    public FanBelt create() { return new FanBelt(); }
  }
}

class GeneratorBelt extends Belt {
  public static class Factory
  implements typeinfo.factory.Factory<GeneratorBelt> {
    public GeneratorBelt create() {
      return new GeneratorBelt();
    }
  }
}

class PowerSteeringBelt extends Belt {
```

```
  public static class Factory
  implements typeinfo.factory.Factory<PowerSteeringBelt> {
    public PowerSteeringBelt create() {
      return new PowerSteeringBelt();
    }
  }
}

public class RegisteredFactories {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      System.out.println(Part.createRandom());
  }
} /* Output:
GeneratorBelt
CabinAirFilter
GeneratorBelt
AirFilter
PowerSteeringBelt
CabinAirFilter
FuelFilter
PowerSteeringBelt
PowerSteeringBelt
FuelFilter
*///:~
```

Non tutte le classi nella gerarchia devono essere istanziate; in questo caso **Filter** e **Belt** sono soltanto classificatori, pertanto non creerete un'istanza per loro, ma esclusivamente per le rispettive sottoclassi. Se una classe deve essere generata da **createRandom()**, conterrà una classe **Factory** interna. L'unico modo per riutilizzare il nome **Factory**, come si può notare nel codice, è qualificando **typeinfo.factory.Factory**.

Sebbene possiate pensare di servirvi di **Collections.addAll()** per incorporare le factory alla lista, il compilatore esprimerà il proprio "dissenso" con un messaggio di avvertimento "generic array creation", segnalando un'operazione non supportata, come vedrete nel prossimo capitolo; quindi, l'autore ha deciso di tornare a chiamare **add()**. Il metodo **createRandom()** seleziona a

caso un oggetto factory da **partFactories** e ne chiama il metodo **create()** per generare un nuovo **Part**.

**Esercizio 14** (4) Un costruttore è un tipo di metodo factory. Modificate **RegisteredFactories.java** in modo tale che, invece di utilizzare una factory esplicita, l'oggetto della classe venga memorizzato in **List**, e **newInstance()** venga utilizzato per generare ogni oggetto.

**Esercizio 15** (4) Implementate un nuovo **PetCreator** utilizzando le factory registrate e modificate la "facciata" di **Pets** in modo che utilizzi questo nuovo modello di creazione invece dei due esistenti. Accertatevi che gli altri esempi che si servono di **Pets.java** funzionino correttamente.

**Esercizio 16** (4) Modificate la gerarchia di **Coffee** (Capitolo 3) per utilizzare le factory registrate.

## Equivalenza di instanceof e Class

Per quanto riguarda la determinazione delle informazioni sui tipi esiste una differenza notevole tra le due forme di **instanceof** (ossia **instanceof** e **isInstance()**, che producono risultati equivalenti) e il confronto diretto degli oggetti di Class. L'esempio seguente evidenzia questa diversità.

```
//: typeinfo/FamilyVsExactType.java
// La differenza tra instanceof e class
package typeinfo;
import static net.mindview.util.Print.*;

class Base {}
class Derived extends Base {}

public class FamilyVsExactType {
  static void test(Object x) {
    print("Testing x of type" + x.getClass());
    print("x instanceof Base " + (x instanceof Base));
    print("x instanceof Derived "+ (x instanceof Derived));
    print("Base.isInstance(x) "+ Base.class.isInstance(x));
    print("Derived.isInstance(x) " +
      Derived.class.isInstance(x));
```

```
    print("x.getClass() == Base.class " +
      (x.getClass() == Base.class));
    print("x.getClass() == Derived.class " +
      (x.getClass() == Derived.class));
    print("x.getClass().equals(Base.class)) "+
      (x.getClass().equals(Base.class)));
    print("x.getClass().equals(Derived.class)) " +
      (x.getClass().equals(Derived.class)));
  }
  public static void main(String[] args) {
    test(new Base());
    test(new Derived());
  }
} /* Output:
Testing x of type class typeinfo.Base
x instanceof Base true
x instanceof Derived false
Base.isInstance(x) true
Derived.isInstance(x) false
x.getClass() == Base.class true
x.getClass() == Derived.class false
x.getClass().equals(Base.class)) true
x.getClass().equals(Derived.class)) false
Testing x of type class typeinfo.Derived
x instanceof Base true
x instanceof Derived true
Base.isInstance(x) true
Derived.isInstance(x) true
x.getClass() == Base.class false
x.getClass() == Derived.class true
x.getClass().equals(Base.class)) false
x.getClass().equals(Derived.class)) true
*///:~
```

Il metodo **test()** effettua il controllo di tipo con i propri argomenti, ricorrendo a entrambe le forme di **instanceof**, quindi ottiene il riferimento **Class** e utilizza **==** ed **equals()** per valutare l'uguaglianza degli oggetti **Class**. Seb-

bene i risultati di **isInstance()** e **instanceof** siano uguali, così come quelli di **equals()** e **==**, i test portano a conclusioni diverse. In conformità al concetto di tipo, **instanceof** si interroga se la classe corrente sia effettivamente quella che sembra o una sua derivata; di contro, se confrontate gli oggetti **Class** effettivi con **==**, non vi ponete alcun problema di ereditarietà: o è il tipo esatto o non lo è.

## Riflessione: informazioni di classe a runtime

Se non conoscete il tipo esatto di un oggetto, RTTI ve lo dirà. Tuttavia, esiste un limite a questa funzionalità: il tipo deve essere noto al momento della compilazione per rilevarlo mediante RTTI e ottenere informazioni effettivamente utili. In altre parole, il compilatore deve sapere tutto delle classi con cui state lavorando.

A prima vista questa non sembra una limitazione, ma supponete di avere assegnato un riferimento a un oggetto che non rientra nell'ambito del vostro programma; in tal caso la classe dell'oggetto non sarà disponibile al vostro programma neppure al momento dell'esecuzione. Per esempio, immaginate di avere ricevuto un file sul vostro disco o tramite una connessione di rete, e che qualcuno vi abbia detto che si tratta di una classe. Dal momento che questa classe si è resa disponibile dopo che il compilatore ha generato il bytecode del vostro programma, come farete per utilizzarla?

In un ambiente di programmazione tradizionale, questa sembrerebbe un'eventualità irreale ma, poiché vi trovate in un ambiente di programmazione ben più ampio, ci sono situazioni in cui potrebbe verificarsi. La prima è la programmazione basata sui componenti, in cui la costruzione di progetti avviene secondo il modello RAD (*Rapid Application Development*) all'interno di un ambiente visuale di sviluppo integrato chiamato IDE (*Integrated Development Enrivonment*), mediante il quale la creazione di un programma avviene spostando icone, che rappresentano i componenti, su un modulo: i componenti sono poi configurati impostando alcuni parametri in fase di programmazione. Questa configurazione richiede che tutti i componenti siano istanziabili, che espongano una parte di loro stessi e che le loro proprietà siano leggibili e modificabili. Inoltre, i componenti che gestiscono gli eventi dell'interfaccia grafica (GUI, *Graphical User Interface*) devono rendere disponibili le informazioni sui metodi appropriati, in modo che l'IDE possa supportare il programmatore nell'overriding di questi metodi di gestione degli eventi. Il meccanismo per rilevare i metodi disponibili e per produrre i nomi di metodo è fornito dalla riflessione, e

per la programmazione dei componenti di base Java mette a disposizione una struttura denominata JavaBeans (si veda il Volme 3, Capitolo 2).

Un'altra situazione che richiede il rilevamento di informazioni sulla classe al momento dell'esecuzione è la necessità di creare ed eseguire gli oggetti su piattaforme remote, via rete, secondo un meccanismo chiamato RMI (*Remote Invocation Method*), che permette la distribuzione degli oggetti di un programma Java su molti computer.

Questa distribuzione può essere giustificata da molte ragioni. Per esempio, quando eseguite attività ad alto tasso di elaborazione, per accelerare le operazioni potreste decidere di suddividere tali attività distribuendone alcune parti su sistemi che stanno operando al minimo della loro potenza. In altri casi, potreste volere disporre il codice che gestisce funzioni particolari (le cosiddette *Business Rules* di un'architettura client/server multilivello) su un determinato computer, che si trasformerà quindi in un repository comune con l'unico compito di descrivere tali azioni, che risulteranno così facilmente modificabili con effetto su tutto il sistema. Tenete presente che questo tipo di sviluppo è decisamente interessante, poiché quel computer avrà soltanto il compito di semplificare le modifiche al software. L'elaborazione distribuita supporta anche hardware specifico che potrebbe rivelarsi molto utile per eseguire operazioni particolari, quali l'inversione di matrici, ma inadeguato o troppo costoso per la programmazione standard.

La classe **Class** supporta il concetto di *riflessione*, in abbinamento alla libreria **java.lang.reflect** che contiene le classi **Field**, **Method** e **Constructor** (ciascuna delle quali implementa l'interfaccia **Member**). Gli oggetti di questo tipo sono creati dalla macchina virtuale JVM in fase di esecuzione per rappresentare il membro corrispondente nella classe sconosciuta. Potete così utilizzare il costruttore per generare nuovi oggetti, i metodi **get()** e **set()** per leggere e modificare i campi associati agli oggetti **Field** e il metodo **invoke()** per chiamare un metodo associato a un oggetto **Method**. Inoltre potete chiamare i metodi di comodo **getFields()**, **getMethods()**, **getConstructors()** ecc., per restituire array di oggetti che rappresentano i campi, i metodi e i costruttori: in proposito, troverete ampie informazioni nella documentazione JDK per la classe **Class**. Pertanto, le informazioni di classe per gli oggetti anonimi possono essere determinate in fase di esecuzione, e niente deve essere necessariamente noto al momento della compilazione.

È importante che vi rendiate conto che la riflessione non ha nulla di magico. Quando utilizzate la riflessione per interagire con un oggetto di tipo sconosciuto, JVM analizzerà l'oggetto e vedrà che appartiene a una classe particolare, come un normale RTTI. Prima di poter essere utilizzato in qual-

siasi modo, l'oggetto **Class** deve essere caricato: di conseguenza, i file **.class** per quel tipo particolare devono essere disponibili alla macchina virtuale, sul computer locale o via rete. Quindi, la vera differenza tra la riflessione e l'RTTI è che con quest'ultima tecnica il compilatore apre ed esamina il file **.class** in fase di compilazione: in altri termini, avete la possibilità di chiamare tutti i metodi di un oggetto nel modo "normale". Con la riflessione, invece, i file **.class** non sono disponibili al momento della compilazione, ma vengono aperti ed esaminati a runtime.

## *Un estrattore dei metodi di classe*

Normalmente non avrete bisogno di utilizzare direttamente le utilità di riflessione, che potranno comunque rivelarsi preziose quando genererete codice più dinamico. La riflessione è insita nel linguaggio per supportare altre funzionalità di Java, quali la serializzazione degli oggetti e JavaBeans, che vedrete nel prosieguo del volume. Tuttavia, vi sono casi in cui è utile poter estrarre dinamicamente le informazioni su una classe.

Considerate un estrattore di metodi di classe: esaminando il codice sorgente della definizione di classe o la documentazione JDK rileverete soltanto i metodi che sono definiti o sovrascritti nell'ambito di quella definizione di classe; potrebbero però essere disponibili decine di altri metodi, che provengono dalle classi di base, la cui individuazione è un'attività lunga e noiosa.[1]

Per fortuna la riflessione consente di scrivere una semplice utility che mostra automaticamente l'intera interfaccia. Osservatene il funzionamento nell'esempio seguente.

```
//: typeinfo/ShowMethods.java
// Utilizzo della riflessione per mostrare tutti i metodi
// di una classe, anche quelli definiti nella classe di base.
// {Args: ShowMethods}
import java.lang.reflect.*;
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class ShowMethods {
```

1. Soprattutto in passato. Sun ha tuttavia migliorato notevolmente la documentazione HTML Java ed è ora più facile vedere i metodi della classe di base.

```
private static String usage =
  "usage:\n" +
  "ShowMethods qualified.class.name\n" +
  "To show all methods in class or:\n" +
  "ShowMethods qualified.class.name word\n" +
  "To search for methods involving 'word'";
private static Pattern p = Pattern.compile("\\w+\\.");
public static void main(String[] args) {
  if(args.length < 1) {
    print(usage);
    System.exit(0);
  }
  int lines = 0;
  try {
    Class<?> c = Class.forName(args[0]);
    Method[] methods = c.getMethods();
    Constructor[] ctors = c.getConstructors();
    if(args.length == 1) {
      for(Method method : methods)
        print(
          p.matcher(method.toString()).replaceAll(""));
      for(Constructor ctor : ctors)
        print(p.matcher(ctor.toString()).replaceAll(""));
      lines = methods.length + ctors.length;
    } else {
      for(Method method : methods)
        if(method.toString().indexOf(args[1]) != -1) {
          print(
            p.matcher(method.toString()).replaceAll(""));
          lines++;
        }
      for(Constructor ctor : ctors)
        if(ctor.toString().indexOf(args[1]) != -1) {
          print(p.matcher(
            ctor.toString()).replaceAll(""));
          lines++;
```

```
        }
      }
    } catch(ClassNotFoundException e) {
      print("No such class: " + e);
    }
  }
} /* Output:
public static void main(String[])
public native int hashCode()
public final native Class getClass()
public final void wait(long,int) throws InterruptedException
public final void wait() throws InterruptedException
public final native void wait(long) throws
InterruptedException
public boolean equals(Object)
public String toString()
public final native void notify()
public final native void notifyAll()
public ShowMethods()
*///:~
```

I metodi di classe **getMethods()** e **getConstructors()** restituiscono rispettivamente un array di **Method** e un array di **Constructor**. Ognuna di queste classi ha altri metodi per analizzare i nomi, gli argomenti e i valori di ritorno dei metodi che rappresentano. Potete tuttavia anche servirvi di **toString()**, come in questo caso, per produrre una **String** con l'intera segnatura del metodo. Il codice rimanente estrae le informazioni dalla riga di comando, determina se una certa segnatura corrisponde alla stringa di destinazione (con **indexOf()**) ed elimina i qualificatori di nome utilizzando le espressioni regolari (si veda il Volume 1, Capitolo 12).

Il risultato prodotto da **Class.forName()** non può essere noto al momento della compilazione, poiché tutte le informazioni sulla segnatura del metodo vengono estratte in sede di esecuzione. Se leggete la documentazione JDK per la riflessione, vedrete che Java fornisce supporto sufficiente per impostare e realizzare una chiamata di metodo su un oggetto sconosciuto al momento della compilazione: nel prosieguo del libro troverete alcuni esempi sull'argomento. Anche se inizialmente penserete di non avere mai bisogno di questa funzionalità, il valore della riflessione completa potrebbe sorprendervi.

L'output dell'esempio precedente è ottenuto con la seguente istruzione da riga di comando:

```
java ShowMethods ShowMethods
```

Potete notare che l'output include un costruttore **public** preordinato, sebbene non sia stato definito alcun costruttore. Il costruttore elencato è quello sintetizzato automaticamente dal compilatore. Se in seguito rendete **ShowMethods** una classe non **public**, ovvero con accesso di tipo package, il costruttore sintetizzato predefinito non apparirà più nell'output. Al costruttore sintetizzato viene automaticamente assegnato lo stesso tipo di accesso della classe.

Un altro interessante esperimento consiste nel chiamare **Java ShowMethods java.lang.String** con un ulteriore argomento **char**, **int**, **String** ecc.

Questa utility può effettivamente farvi risparmiare una notevole quantità di tempo in fase di programmazione, quando non ricordate se una classe possiede un determinato metodo e non volete perdervi nei meandri dell'indice o della gerarchia di classi della documentazione JDK, o quando non sapete se una classe può gestire oggetti particolari.

Il Volume 3, Capitolo 2 contiene una versione GUI di questo programma, personalizzata per l'estrazione di informazioni sui componenti Swing, che potrete lasciare attiva mentre programmate, per eseguire rapidi controlli.

**Esercizio 17** (2) Modificate l'espressione regolare in **ShowMethods.java** per eliminare anche le parole chiave **native** e **final**. Suggerimento: utilizzate l'operatore OR "|".

**Esercizio 18** (1) Trasformate **ShowMethods** in classe non **public** e verificate che il costruttore sintetizzato predefinito non appaia più nell'output.

**Esercizio 19** (4) In **ToyTest.java**, servitevi della riflessione per generare un oggetto **Toy** utilizzando un costruttore non predefinito.

**Esercizio 20** (5) Leggete la documentazione JDK per l'interfaccia **java.lang.Class**, scaricabile all'indirizzo *http://java.sun.com*. Scrivete un programma che accetti il nome di una classe come argomento da riga di comando, poi utilizzate i metodi **Class** per ottenere tutte le informazioni disponibili per quella classe. Testate il programma con una classe della libreria standard e una che creerete.

## Proxy dinamici

*Proxy* è uno dei design pattern fondamentali: è un oggetto che potete inserire in luogo di quello "vero" per offrire funzionalità aggiuntive o differenti. Di norma i proxy sono utilizzati nella comunicazione con gli oggetti "reali", pertanto agiscono, di fatto, da intermediari. Segue un esempio banale della struttura di un proxy.

```
//: typeinfo/SimpleProxyDemo.java
import static net.mindview.util.Print.*;

interface Interface {
  void doSomething();
  void somethingElse(String arg);
}

class RealObject implements Interface {
  public void doSomething() { print("doSomething"); }
  public void somethingElse(String arg) {
    print("somethingElse " + arg);
  }
}

class SimpleProxy implements Interface {
  private Interface proxied;
  public SimpleProxy(Interface proxied) {
    this.proxied = proxied;
  }
  public void doSomething() {
    print("SimpleProxy doSomething");
    proxied.doSomething();
  }
  public void somethingElse(String arg) {
    print("SimpleProxy somethingElse " + arg);
    proxied.somethingElse(arg);
  }
}
```

```
class SimpleProxyDemo {
  public static void consumer(Interface iface) {
    iface.doSomething();
    iface.somethingElse("bonobo");
  }
  public static void main(String[] args) {
    consumer(new RealObject());
    consumer(new SimpleProxy(new RealObject()));
  }
} /* Output:
doSomething
somethingElse bonobo
SimpleProxy doSomething
doSomething
SimpleProxy somethingElse bonobo
somethingElse bonobo
*///:~
```

Poiché **consumer()** accetta una **Interface**, non può sapere se sta ottenendo un vero oggetto (**RealObject**) o un **SimpleProxy**, dal momento che entrambi implementano **Interface**. In realtà, il **SimpleProxy**, che si trova tra il client e l'oggetto **RealObject**, esegue le operazioni e poi chiama l'identico metodo su un **RealObject**.

Un proxy può essere utile in qualunque momento vogliate segregare le operazioni aggiuntive in una posizione diversa da quella in cui si trova l'"oggetto reale", soprattutto quando volete disporre della massima libertà nel passare dalla condizione di "non utilizzo" delle operazioni supplementari alla condizione di "utilizzo" e viceversa: tenete presente che il motivo che legittima il ricorso ai design pattern è l'incapsulamento delle modifiche, pertanto dovete cambiare qualcosa per giustificare l'utilizzo del pattern. Per esempio, supponete di volere monitorare le chiamate ai metodi nel **RealObject** o misurare il carico di sistema che tali chiamate comportano. Questo non è certo il tipo di codice che vorrete includere nella vostra applicazione, quindi un proxy vi consentirà di aggiungerlo o eliminarlo con la massima facilità.

I *proxy dinamici* di Java sono un'ulteriore evoluzione del concetto di "intermediario": essi generano l'oggetto proxy e gestiscono le chiamate ai metodi sottoposti a proxy in modo assolutamente dinamico. Tutte le chiamate eseguite su un proxy dinamico vengono ridirette a un unico gestore di chiamate

(*invocation handler*), che ha il compito di determinare il tipo di chiamata e decidere il da farsi. L'esempio che segue è il codice di **SimpleProxyDemo.java** riscritto per utilizzare un proxy dinamico.

```
//: typeinfo/SimpleDynamicProxy.java
import java.lang.reflect.*;

class DynamicProxyHandler implements InvocationHandler {
  private Object proxied;
  public DynamicProxyHandler(Object proxied) {
    this.proxied = proxied;
  }
  public Object
  invoke(Object proxy, Method method, Object[] args)
  throws Throwable {
    System.out.println("**** proxy: " + proxy.getClass() +
      ", method: " + method + ", args: " + args);
    if(args != null)
      for(Object arg : args)
        System.out.println("  " + arg);
    return method.invoke(proxied, args);
  }
}

class SimpleDynamicProxy {
  public static void consumer(Interface iface) {
    iface.doSomething();
    iface.somethingElse("bonobo");
  }
  public static void main(String[] args) {
    RealObject real = new RealObject();
    consumer(real);
    // Inserisce un proxy ed esegue ancora la chiamata
    Interface proxy = (Interface)Proxy.newProxyInstance(
      Interface.class.getClassLoader(),
      new Class[]{ Interface.class },
      new DynamicProxyHandler(real));
```

```
    consumer(proxy);
  }
} /* Output: (95% match)
doSomething
somethingElse bonobo
**** proxy: class $Proxy0, method: public abstract void
Interface.doSomething(), args: null
doSomething
**** proxy: class $Proxy0, method: public abstract void
Interface.somethingElse(java.lang.String), args:
java.lang.Object;@5afd29
  bonobo
somethingElse bonobo
*///:~
```

Viene creato un proxy dinamico chiamando il metodo **static Proxy.newProxyInstance()**, che richiede un caricatore di classe (di solito è sufficiente quello di un oggetto che sia già stato caricato), un elenco di interfacce (non classi o classi astratte) che volete siano implementate dal proxy e un'implementazione dell'interfaccia **InvocationHandler**. Il proxy dinamico redirigerà tutte le chiamate al gestore delle chiamate (**InvocationHandler**), pertanto al costruttore di tale gestore viene generalmente dato il riferimento all'oggetto "reale", in modo che il gestore stesso possa inoltrare le richieste una volta eseguito il suo compito di intermediazione.

Al metodo **invoke()** viene passato l'oggetto proxy, qualora abbiate necessità di distinguere la provenienza delle richieste; in molti casi, però, tale distinzione non è indispensabile. Tuttavia, prestate attenzione quando chiamate i metodi sul proxy all'interno di **invoke()**, poiché le chiamate attraverso l'interfaccia sono ridirette al proxy.

In generale eseguirete l'operazione tramite il proxy e soltanto dopo utilizzerete **Method.invoke()** per trasmettere la richiesta all'oggetto di competenza, passando gli argomenti necessari. Questa tecnica può apparire limitativa, come se poteste eseguire unicamente operazioni generiche. Se non altro, tuttavia, potrete filtrare determinate chiamate di metodo, trascurando le altre.

```
//: typeinfo/SelectingMethods.java
// Ricerca di metodi particolari in un proxy dinamico.
import java.lang.reflect.*;
import static net.mindview.util.Print.*;
```

```
class MethodSelector implements InvocationHandler {
  private Object proxied;
  public MethodSelector(Object proxied) {
    this.proxied = proxied;
  }
  public Object
  invoke(Object proxy, Method method, Object[] args)
  throws Throwable {
    if(method.getName().equals("interesting"))
      print("Proxy detected the interesting method");
    return method.invoke(proxied, args);
  }
}

interface SomeMethods {
  void boring1();
  void boring2();
  void interesting(String arg);
  void boring3();
}

class Implementation implements SomeMethods {
  public void boring1() { print("boring1"); }
  public void boring2() { print("boring2"); }
  public void interesting(String arg) {
    print("interesting " + arg);
  }
  public void boring3() { print("boring3"); }
}

class SelectingMethods {
  public static void main(String[] args) {
    SomeMethods proxy= (SomeMethods)Proxy.newProxyInstance(
      SomeMethods.class.getClassLoader(),
      new Class[]{ SomeMethods.class },
      new MethodSelector(new Implementation()));
    proxy.boring1();
```

```
      proxy.boring2();
      proxy.interesting("bonobo");
      proxy.boring3();
    }
  } /* Output:
  boring1
  boring2
  Proxy detected the interesting method
  interesting bonobo
  boring3
  *///:~
```

In questo esempio il codice si limita a cercare i nomi di metodo, ma potreste anche ampliare la ricerca ad altri aspetti della segnatura di metodo, nonché cercare determinati valori di argomento.

Il proxy dinamico non è certo uno strumento di utilizzo quotidiano, tuttavia può risolvere alcuni tipi di problemi in modo efficace. Troverete maggiori informazioni su Proxy e altri design pattern in *Thinking in Patterns* (*www.mindview.net*) e in *Design Patterns*, di Erich Gamma *et al.* (Addison-Wesley, 1995).

**Esercizio 21** (3) Modificate **SimpleProxyDemo.java** in modo che misuri i tempi delle chiamate di metodo.

**Esercizio 22** (3) Modificate **SimpleDynamicProxy.java** in modo che misuri i tempi delle chiamate di metodo.

**Esercizio 23** (3) In **SimpleDynamicProxy.java**, all'interno di **invoke()** provate a visualizzare gli argomenti del proxy e illustrate che cosa accade.

**Progetto** Scrivete un'applicazione che utilizzi i proxy dinamici per implementare le *transazioni*, nella quale il proxy esegua una *commit* se la chiamata sottoposta al proxy ha avuto successo (senza sollevare eccezioni) e un *rollback* in caso di fallimento. Le operazioni di *commit* e *rollback* dovranno lavorare con un file di testo esterno, non soggetto al meccanismo di eccezioni Java. Dovrete prestare attenzione all'*atomicità* delle operazioni.[2]

2. I progetti sono proposte da utilizzare, per esempio, come pianificazioni a termine. Le soluzioni ai progetti non sono incluse nella guida delle soluzioni agli esercizi.

# Oggetti null

Quando utilizzate il valore **null** nativo per indicare l'assenza di un oggetto, prima di utilizzarlo dovete verificare che il riferimento non sia **null**. Questa operazione può risultare noiosa e dare luogo a una notevole quantità di codice. Il problema è che **null** non ha alcun comportamento in sé, a parte quello di produrre una **NullPointerException** qualora proviate a servirvene. A volte può essere utile implementare il concetto di un oggetto null (*Null Object*) che accetti messaggi per l'oggetto che "sostituisce", ma restituisca valori che indicano l'assenza di oggetti "reali" effettivi.[3]

In questo modo potete dare per scontato che tutti gli oggetti sono validi e non dovete perdere tempo per analizzare il codice e controllare la presenza di un valore **null**.

Nonostante possa essere divertente immaginare un linguaggio di programmazione che crei in modo automatico gli oggetti null, nella pratica non ha senso utilizzarli indiscriminatamente. Talvolta è opportuno verificare i valori **null**, a volte potreste ragionevolmente aspettarvi di non incontrarne, altre ancora potreste accettare che venga sollevata una **NullPointerException**. Il punto in cui sembrerebbe più utile inserire gli oggetti null è "il più vicino possibile ai dati", con oggetti che rappresentano entità nello spazio del problema. Come esempio considerate il fatto che molti sistemi hanno una classe **Person** anche se in alcuni casi non esiste una persona reale (oppure esiste, ma non si hanno ancora tutte le necessarie informazioni su di essa), per cui di solito utilizzerete un riferimento **null**, valutandone la presenza. In questi casi, come alternativa potrete realizzare un oggetto null. Anche se tale oggetto rispondesse a tutti i messaggi cui risponderebbe l'"oggetto reale", vi servirà tuttavia un modo per testare la condizione di nullità. A questo scopo, l'approccio più semplice consiste nel generare un'interfaccia di *tagging*.

```
//: net/mindview/util/Null.java
package net.mindview.util;
public interface Null {} ///:~
```

Questo consente all'operatore **instanceof** di rilevare l'oggetto null e, soprattutto, non richiede di integrare in tutte le classi un metodo **isNull()** che, in

3. Scoperto da Bobby Woolf e Bruce Anderson, il concetto di *Null Object* può essere considerato come un caso speciale del design pattern *Strategy*. Una variante è il pattern *Null Iterator*, che fa in modo che l'iterazione dei nodi di una gerarchia composita sia trasparente nei confronti del client: in tal modo il client potrà adottare la stessa logica per iterare i nodi compositi e dell'alberatura.

realtà, non sarebbe nient'altro che un modo diverso di eseguire l'RTTI: dal momento che esiste, perché non utilizzare la funzionalità nativa?

```
//: typeinfo/Person.java
// Una classe con un oggetto null (Null Object).
import net.mindview.util.*;

class Person {
  public final String first;
  public final String last;
  public final String address;
  // ecc.
  public Person(String first, String last, String address){
    this.first = first;
    this.last = last;
    this.address = address;
  }
  public String toString() {
    return "Person: " + first + " " + last + " " + address;
  }
  public static class NullPerson
  extends Person implements Null {
    private NullPerson() { super("None", "None", "None"); }
    public String toString() { return "NullPerson"; }
  }
  public static final Person NULL = new NullPerson();
} ///:~
```

In generale, l'oggetto null sarà un pattern *Singleton*, che in questo caso viene creato come istanza **static final**. Questo accorgimento funziona perché **Person** è *immutabile*: potete solo impostare i valori nel costruttore e poi leggerli, ma non modificarli, poiché le **String** sono intrinsecamente immutabili. In pratica, se desiderate cambiare **NullPerson**, potrete soltanto sostituirlo con un nuovo oggetto **Person**. Tenete presente che è possibile rilevare il **Null** generico oppure il più specifico **NullPerson** utilizzando **instanceof**, ma con il pattern Singleton potete anche utilizzare soltanto il metodo **equals()** oppure **==** per confrontarlo con **Person.NULL**.

Ora supponete di vivere nei gloriosi giorni della New Economy, e di disporre di fondi illimitati per il vostro "meraviglioso progetto". Siete pronti per assumere moltissimo personale, e mentre procedete alle valutazioni delle diverse posizioni che avete previsto potete utilizzare gli oggetti null per **Person** come segnaposto per ogni **Position**.

```
//: typeinfo/Position.java

class Position {
  private String title;
  private Person person;
  public Position(String jobTitle, Person employee) {
    title = jobTitle;
    person = employee;
    if(person == null)
      person = Person.NULL;
  }
  public Position(String jobTitle) {
    title = jobTitle;
    person = Person.NULL;
  }
  public String getTitle() { return title; }
  public void setTitle(String newTitle) {
    title = newTitle;
  }
  public Person getPerson() { return person; }
  public void setPerson(Person newPerson) {
    person = newPerson;
    if(person == null)
      person = Person.NULL;
  }
  public String toString() {
    return "Position: " + title + " " + person;
  }
} ///:~
```

Con **Position** non dovete necessariamente creare un oggetto null, poiché l'esistenza di **Person.NULL** implica una **Position** di tipo **null**. Tenete presente che in seguito potreste riscontrare la necessità di aggiungere un oggetto null esplicito per **Position**: in ogni caso, l'approccio YAGNI (*You Aren't Going to Need It,* cioè "Almeno per ora non vi serve") raccomanda di testare la soluzione più semplice come prima bozza di codice, aspettando di verificare se altri aspetti del programma richiederanno questa funzionalità, invece di dare per scontato che sia necessaria da subito.[4]

Ora la classe **Staff** può cercare gli oggetti null ogniqualvolta registrate le assunzioni.

```
//: typeinfo/Staff.java
import java.util.*;

public class Staff extends ArrayList<Position> {
  public void add(String title, Person person) {
    add(new Position(title, person));
  }
  public void add(String... titles) {
    for(String title : titles)
      add(new Position(title));
  }
  public Staff(String... titles) { add(titles); }
  public boolean positionAvailable(String title) {
    for(Position position : this)
      if(position.getTitle().equals(title) &&
         position.getPerson() == Person.NULL)
        return true;
    return false;
  }
  public void fillPosition(String title, Person hire) {
    for(Position position : this)
      if(position.getTitle().equals(title) &&
         position.getPerson() == Person.NULL) {
```

4. Un dogma della programmazione estrema (XP, *Extreme Programming*), come *"Do the simplest thing that could possibly work"*, ossia "Fate la cosa più semplice che potrebbe funzionare".

```
        position.setPerson(hire);
        return;
      }
    throw new RuntimeException(
      "Position " + title + " not avaliable");
  }
  public static void main(String[] args) {
    Staff staff = new Staff("President", "CTO",
      "Marketing Manager", "Product Manager",
      "Project Lead", "Software Engineer",
      "Software Engineer", "Software Engineer",
      "Software Engineer", "Test Engineer",
      "Technical Writer");
    staff.fillPosition("President",
      new Person("Me", "Last", "The Top, Lonely At"));
    staff.fillPosition("Project Lead",
      new Person("Janet", "Planner", "The Burbs"));
    if(staff.positionAvailable("Software Engineer"))
      staff.fillPosition("Software Engineer",
        new Person("Bob", "Coder", "Bright Light City"));
    System.out.println(staff);
  }
} /* Output:
[Position: President Person: Me Last The Top, Lonely At,
Position: CTO NullPerson, Position: Marketing Manager
NullPerson, Position: Product Manager NullPerson, Position:
Project Lead Person: Janet Planner The Burbs, Position:
Software Engineer Person: Bob Coder Bright Light City,
Position: Software Engineer NullPerson, Position: Software
Engineer NullPerson, Position: Software Engineer
NullPerson, Position: Test Engineer NullPerson, Position:
Technical Writer NullPerson]
*///:~
```

Notate che dovete comunque testare la presenza di oggetti null in qualche punto del codice, il che non è molto differente dal controllare **null**. Tuttavia, in altri punti del codice come, in questo caso, nelle conversioni **toString()**,

non avrete più la necessità di eseguire test aggiuntivi poiché potrete dare per scontato che tutti i riferimenti agli oggetti siano validi.

Se lavorate con le interfacce anziché con classi concrete, potrete utilizzare un **DynamicProxy** per generare automaticamente gli oggetti null. Immaginate di avere un'interfaccia **Robot** che definisca un nome, il modello e una **List<Operation>** con l'indicazione delle capacità operative del **Robot**. La classe **Operation** conterrà la descrizione e un comando di tipo pattern *Command*.

```
//: typeinfo/Operation.java

public interface Operation {
  String description();
  void command();
} ///:~
```

Per accedere ai servizi di **Robot** potete chiamare il metodo **operations()**.

```
//: typeinfo/Robot.java
import java.util.*;
import net.mindview.util.*;

public interface Robot {
  String name();
  String model();
  List<Operation> operations();
  class Test {
    public static void test(Robot r) {
      if(r instanceof Null)
        System.out.println("[Null Robot]");
      System.out.println("Robot name: " + r.name());
      System.out.println("Robot model: " + r.model());
      for(Operation operation : r.operations()) {
        System.out.println(operation.description());
        operation.command();
      }
    }
  }
} ///:~
```

Questo codice include anche una classe nidificata che si occupa dei test.

A questo punto potete creare un **Robot** spazzaneve.

```
//: typeinfo/SnowRemovalRobot.java
import java.util.*;

public class SnowRemovalRobot implements Robot {
  private String name;
  public SnowRemovalRobot(String name) {this.name = name;}
public String name() { return name; }
  public String model() { return "SnowBot Series 11"; }
  public List<Operation> operations() {
    return Arrays.asList(
      new Operation() {
        public String description() {
          return name + " can shovel snow ";
        }
        public void command() {
          System.out.println(name + " shoveling snow");
        }
      },
      new Operation() {
        public String description() {
          return name + " can chip ice";
        }
        public void command() {
          System.out.println(name + " chipping ice");
        }
      },
      new Operation() {
        public String description() {
          return name + " can clear the roof";
        }
        public void command() {
          System.out.println(name + " clearing roof");
        }
```

```
      }
    );
  }
  public static void main(String[] args) {
    Robot.Test.test(new SnowRemovalRobot("Slusher"));
  }
} /* Output:
Robot name: Slusher
Robot model: SnowBot Series 11
Slusher can shovel snow
Slusher shoveling snow
Slusher can chip ice
Slusher chipping ice
Slusher can clear the roof
Slusher clearing roof
*///:~
```

Probabilmente esisteranno molti tipi di **Robot** e vorrete che per ogni tipo ciascun oggetto null esegua un'azione particolare; in questo caso specifico, integrare informazioni sul tipo esatto di **Robot** cui corrisponde l'oggetto null. Queste informazioni saranno intercettate dal proxy dinamico.

```
//: typeinfo/NullRobot.java
// Utilizzo di un proxy dinamico per creare un oggetto null.
import java.lang.reflect.*;
import java.util.*;
import net.mindview.util.*;

class NullRobotProxyHandler implements InvocationHandler {
  private String nullName;
  private Robot proxied = new NRobot();
  NullRobotProxyHandler(Class<? extends Robot> type) {
    nullName = type.getSimpleName() + " NullRobot";
  }
  private class NRobot implements Null, Robot {
    public String name() { return nullName; }
    public String model() { return nullName; }
```

```
    public List<Operation> operations() {
      return Collections.emptyList();
    }
  }
  public Object
  invoke(Object proxy, Method method, Object[] args)
  throws Throwable {
    return method.invoke(proxied, args);
  }
}

public class NullRobot {
  public static Robot
  newNullRobot(Class<? extends Robot> type) {
    return (Robot)Proxy.newProxyInstance(
      NullRobot.class.getClassLoader(),
      new Class[]{ Null.class, Robot.class },
      new NullRobotProxyHandler(type));
  }
  public static void main(String[] args) {
    Robot[] bots = {
      new SnowRemovalRobot("SnowBee"),
      newNullRobot(SnowRemovalRobot.class)
    };
    for(Robot bot : bots)
      Robot.Test.test(bot);
  }
} /* Output:
Robot name: SnowBee
Robot model: SnowBot Series 11
SnowBee can shovel snow
SnowBee shoveling snow
SnowBee can chip ice
SnowBee chipping ice
SnowBee can clear the roof
SnowBee clearing roof
```

```
[Null Robot]
Robot name: SnowRemovalRobot NullRobot
Robot model: SnowRemovalRobot NullRobot
*///:~
```

Ogni volta che avete bisogno di un oggetto **Robot** di tipo null, dovete chiamare il metodo **newNullRobot()**, passando come parametro il tipo di **Robot** per cui volete attivare il proxy. Il proxy soddisfa le necessità delle interfacce **Null** e **Robot** e fornisce il nome e il tipo specifici su cui opera.

### Oggetti mock e stub

Evoluzioni logiche dell'oggetto null sono gli oggetti *mock* (letteralmente, "oggetti finti") e gli *stub*. Come gli oggetti null, entrambi sono sostitutivi degli oggetti "reali" che saranno utilizzati nella versione finale del programma. Tuttavia, sia gli oggetti mock sia gli stub "fingono" di essere oggetti esistenti che trasmettono informazioni reali, anziché svolgere il ruolo di segnaposto "intelligenti" per il valore **null**, come gli oggetti null.

La differenza tra oggetti mock e stub è la seguente: generalmente gli oggetti finti sono "leggeri" e autovalutativi, e molti di essi vengono generati soltanto per gestire situazioni difficili; di contro, gli stub sono più "pesanti" e spesso riutilizzati dopo ogni valutazione. Gli stub possono essere configurati per assumere un comportamento diverso in funzione del modo in cui vengono chiamati: uno stub, quindi, è un oggetto specializzato che può eseguire molte operazioni, a differenza degli oggetti mock, più semplici, che di solito vengono utilizzati in quantità.

**Esercizio 24** (4) Aggiungete oggetti null a **RegisteredFactories.java**.

## Interfacce e informazioni di tipo

Un obiettivo importante della parola chiave **interface** è consentire al programmatore di isolare i componenti, riducendo così il cosiddetto "accoppiamento" (*coupling*). Scrivendo codice che utilizza le interfacce rispettate esattamente questo obiettivo, tuttavia con le informazioni di tipo è possibile aggirare questa regola: le interfacce non costituiscono un'assoluta garanzia di disaccoppiamento. Di seguito è mostrato un esempio, che inizia con un'interfaccia.

```
//: typeinfo/interfacea/A.java
package typeinfo.interfacea;
```

```
public interface A {
  void f();
} ///:~
```

Questa interfaccia viene poi implementata; osservate come sia possibile "ingannare" il tipo reale di implementazione.

```
//: typeinfo/InterfaceViolation.java
// Come aggirare un'interfaccia.
import typeinfo.interfacea.*;

class B implements A {
  public void f() {}
  public void g() {}
}

public class InterfaceViolation {
  public static void main(String[] args) {
    A a = new B();
    a.f();
    // a.g(); // Errore di compilazione
    System.out.println(a.getClass().getName());
    if(a instanceof B) {
      B b = (B)a;
      b.g();
    }
  }
} /* Output:
B
*///:~
```

Utilizzando RTTI scoprirete che **a** è stato implementato come **B**. Eseguendo un casting a **B**, potete chiamare un metodo che non è in **A**.

Questo è perfettamente legale e accettabile, ma potreste volere far sì che i programmatori client non eseguano questa operazione, poiché darebbe loro l'occasione di legarsi al vostro codice più di quanto potreste desiderare: in pratica, potreste pensare che la parola chiave **interface** vi protegga, ma non

è così, e il fatto che stiate utilizzando **B** per implementare **A** diventerà di dominio pubblico.[5]

Una soluzione consiste semplicemente nel prendere atto che i programmatori sono liberi di fare ciò che desiderano, qualora decidano di utilizzare la classe reale invece dell'interfaccia. Questo approccio potrà risultare ragionevole in molti casi, ma se non lo consideraste sufficiente potreste volere applicare controlli più rigidi.

L'approccio più semplice consiste nel ricorrere al package access per l'implementazione, in modo che i client esterni al package non possano accedervi.

```
//: typeinfo/packageaccess/HiddenC.java
package typeinfo.packageaccess;
import typeinfo.interfacea.*;
import static net.mindview.util.Print.*;

class C implements A {
  public void f() { print("public C.f()"); }
  public void g() { print("public C.g()"); }
  void u() { print("package C.u()"); }
  protected void v() { print("protected C.v()"); }
  private void w() { print("private C.w()"); }
}

public class HiddenC {
  public static A makeA() { return new C(); }
} ///:~
```

L'unica parte **public** di questo package, **HiddenC**, alla chiamata genera un'interfaccia **A**. Ciò che è interessante in questa tecnica è che anche se **makeA()** avesse restituito **C**, esternamente si sarebbe comunque potuto utilizzare soltanto **A**, poiché non è possibile chiamare **C** dall'esterno del pacchetto.

A questo punto, se tentate di eseguire il downcast a **C** non potrete farlo, dal momento che il tipo "**C**" non è disponibile all'esterno del package.

5. Il caso più famoso ha interessato il sistema operativo Windows che disponeva di una serie di funzionalità API utilizzabili e di un insieme di funzioni API non pubbliche, ma visibili, che era possibile scoprire e utilizzare. Per risolvere molti problemi, i programmatori hanno scelto di utilizzare queste ultime. Microsoft è stata quindi obbligata a gestirle come se facessero parte delle funzioni API pubbliche, il che naturalmente ha comportato dispendio di risorse e costi.

```
//: typeinfo/HiddenImplementation.java
// Come aggirare l'accesso di pacchetto.
import typeinfo.interfacea.*;
import typeinfo.packageaccess.*;
import java.lang.reflect.*;

public class HiddenImplementation {
  public static void main(String[] args) throws Exception {
    A a = HiddenC.makeA();
    a.f();
    System.out.println(a.getClass().getName());
    // Errore di compilazione: cannot find symbol 'C':
    /* if(a instanceof C) {
      C c = (C)a;
      c.g();
    } */
    // Sorpresa! La riflessione permette ancora di chiamare g():
    callHiddenMethod(a, "g");
    // e persino i metodi meno accessibili!
    callHiddenMethod(a, "u");
    callHiddenMethod(a, "v");
    callHiddenMethod(a, "w");
  }
  static void callHiddenMethod(Object a, String methodName)
  throws Exception {
    Method g = a.getClass().getDeclaredMethod(methodName);
    g.setAccessible(true);
    g.invoke(a);
  }
} /* Output:
public C.f()
typeinfo.packageaccess.C
public C.g()
package C.u()
protected C.v()
private C.w()
*///:~
```

Come potete vedere, utilizzando la riflessione è ancora possibile raggiungere e chiamare tutti i metodi, persino quelli **private**! Se conoscete il nome del metodo, potete chiamare **setAccessible(true)** sull'oggetto **Method** per rendere accessibile tale metodo, come potete vedere in **callHiddenMethod()**.

Potreste allora pensare di impedire questo comportamento distribuendo unicamente il codice compilato, ma anche questa non è una soluzione; in tal caso, infatti, sarebbe sufficiente utilizzare **javap**, il decompilatore fornito con il JDK. Il comando da eseguire è il seguente.

```
javap -private C
```

L'opzione **-private** indica di visualizzare tutti i membri, anche quelli riservati. Quello che segue è l'output prodotto.

```
class typeinfo.packageaccess.C extends
java.lang.Object implements typeinfo.interfacea.A {
    typeinfo.packageaccess.C();
    public void f();
    public void g();
    void u();
    protected void v();
    private void w();
}
```

In questo modo, chiunque può ottenere i nomi e le segnature dei vostri metodi **private** e chiamarli.

E se implementaste l'interfaccia come classe interna **private**? Segue un esempio di questa implementazione.

```
//: typeinfo/InnerImplementation.java
// Le classi interne private non possono nascondersi
// dalla riflessione.
import typeinfo.interfacea.*;
import static net.mindview.util.Print.*;

class InnerA {
  private static class C implements A {
    public void f() { print("public C.f()"); }
```

```
    public void g() { print("public C.g()"); }
    void u() { print("package C.u()"); }
    protected void v() { print("protected C.v()"); }
    private void w() { print("private C.w()"); }
  }
  public static A makeA() { return new C(); }
}

public class InnerImplementation {
  public static void main(String[] args) throws Exception {
    A a = InnerA.makeA();
    a.f();
    System.out.println(a.getClass().getName());
    // La riflessione consente ancora di accedere alla classe
    // private:
    HiddenImplementation.callHiddenMethod(a, "g");
    HiddenImplementation.callHiddenMethod(a, "u");
    HiddenImplementation.callHiddenMethod(a, "v");
    HiddenImplementation.callHiddenMethod(a, "w");
  }
} /* Output:
public C.f()
InnerA$C
public C.g()
package C.u()
protected C.v()
private C.w()
*///:~
```

Anche questo meccanismo non nasconde alcunché alla riflessione. E se provaste con una classe anonima?

```
//: typeinfo/AnonymousImplementation.java
// Le classi interne anonime non possono nascondersi
// dalla riflessione.
import typeinfo.interfacea.*;
import static net.mindview.util.Print.*;
```

```
class Anonymoutilizza {
  public static A makeA() {
    return new A() {
      public void f() { print("public C.f()"); }
      public void g() { print("public C.g()"); }
      void u() { print("package C.u()"); }
      protected void v() { print("protected C.v()"); }
      private void w() { print("private C.w()"); }
    };
  }
}

public class AnonymousImplementation {
  public static void main(String[] args) throws Exception {
    A a = AnonymousA.makeA();
    a.f();
    System.out.println(a.getClass().getName());
    // La riflessione consente ancora di accedere alla classe
    // anonima:
    HiddenImplementation.callHiddenMethod(a, "g");
    HiddenImplementation.callHiddenMethod(a, "u");
    HiddenImplementation.callHiddenMethod(a, "v");
    HiddenImplementation.callHiddenMethod(a, "w");
  }
} /* Output:
public C.f()
AnonymousA$1
public C.g()
package C.u()
protected C.v()
private C.w()
*///:~
```

Sembra che non vi sia modo per evitare che la riflessione acceda e chiami i metodi con accesso non **public**. Questo è vero anche per i campi, sebbene **private**.

```
//: typeinfo/ModifyingPrivateFields.java
import java.lang.reflect.*;

class WithPrivateFinalField {
  private int i = 1;
  private final String s = "I'm totally safe";
  private String s2 = "Am I safe?";
  public String toString() {
    return "i = " + i + ", " + s + ", " + s2;
  }
}

public class ModifyingPrivateFields {
  public static void main(String[] args) throws Exception {
    WithPrivateFinalField pf = new WithPrivateFinalField();
    System.out.println(pf);
    Field f = pf.getClass().getDeclaredField("i");
    f.setAccessible(true);
    System.out.println("f.getInt(pf): " + f.getInt(pf));
    f.setInt(pf, 47);
    System.out.println(pf);
    f = pf.getClass().getDeclaredField("s");
    f.setAccessible(true);
    System.out.println("f.get(pf): " + f.get(pf));
    f.set(pf, "No, you're not!");
    System.out.println(pf);
    f = pf.getClass().getDeclaredField("s2");
    f.setAccessible(true);
    System.out.println("f.get(pf): " + f.get(pf));
    f.set(pf, "No, you're not!");
    System.out.println(pf);
  }
} /* Output:
i = 1, I'm totally safe, Am I safe?
f.getInt(pf): 1
i = 47, I'm totally safe, Am I safe?
```

```
f.get(pf): I'm totally safe
i = 47, I'm totally safe, Am I safe?
f.get(pf): Am I safe?
i = 47, I'm totally safe, No, you're not!
*///:~
```

Tuttavia, i campi **final** sono effettivamente al riparo da modifiche. Il runtime di Java accetta qualsiasi tentativo di cambiamento senza opporre resistenza, tuttavia non accade alcunché.

In generale, tuttavia, tutte queste violazioni di accesso non sono il peggior problema del mondo. Chi applicasse una simile tecnica per chiamare metodi che avete contrassegnato come **private** o con accesso di tipo package, indicando così in modo inequivocabile che non dovrebbero essere chiamati, ben difficilmente potrebbe protestare nel momento in cui cambiaste una funzionalità di quei metodi. D'altra parte, il fatto di avere sempre a disposizione un "accesso di servizio" a una classe, la cosiddetta *backdoor*, può permettervi di fare fronte a determinati problemi che diversamente potrebbero essere difficili o impossibili da risolvere: come vedete, i vantaggi della riflessione in generale sono innegabili.

**Esercizio 25** (2) Create una classe contenente metodi **private**, **protected** e con accesso di tipo di package, quindi scrivete il codice per accedere a questi metodi dall'esterno del pacchetto della classe.

## Riepilogo

La funzionalità RTTI consente di scoprire le informazioni sui tipi da un riferimento anonimo alla classe di base. Con questi presupposti è possibile che venga utilizzato malamente dai programmatori principianti, considerato che potrebbe avere senso utilizzarlo prima delle chiamate di metodo polimorfiche. I lettori che hanno esperienza nella programmazione procedurale potrebbero trovare difficile non organizzare i programmi sotto forma di istruzioni **switch**. Potete farlo ricorrendo all'RTTI, perdendo tuttavia il notevole valore aggiunto che il polimorfismo conferisce allo sviluppo e alla manutenzione del codice. Ricordate che l'obiettivo della programmazione OOP è utilizzare chiamate di metodo polimorfiche ovunque sia possibile, e la funzionalità RTTI solo quando sia indispensabile.

Tuttavia l'utilizzo di chiamate di metodo polimorfiche, come dovrebbero essere intese, richiede il controllo della definizione della classe di base, poiché a un certo stadio dello sviluppo del programma potreste scoprire che la classe

di base non include il metodo che vi occorre. Se la classe di base proviene da una libreria fornita da altri, una soluzione è l'RTTI: potrete ereditare un nuovo tipo e aggiungere il vostro nuovo metodo. In altri punti del codice potrete rilevare il vostro tipo specifico e chiamare il metodo speciale che avete aggiunto. Questa tecnica non annulla il polimorfismo né l'estensibilità del programma, poiché l'aggiunta di un nuovo tipo non vi obbligherà a ricercare tutte le istruzioni **switch** presenti nel programma stesso. Tuttavia, quando aggiungerete codice che si serve della vostra nuova funzionalità, dovrete servirvi di RTTI per individuare il vostro tipo particolare.

L'inserimento di una funzionalità in una classe di base potrebbe implicare che, a vantaggio di una determinata classe, tutte le altre classi da essa derivate richiedano l'utilizzo di qualche insignificante metodo stub. Questo rende meno chiara l'interfaccia, e infastidisce chi deve sovrascrivere i metodi astratti quando derivano dalla classe di base in questione. Considerate, per esempio, una gerarchia di classi che rappresenta gli strumenti musicali. Supponete di volere regolare le valvole di sfiato di tutti gli ottoni della vostra orchestra: una possibilità potrebbe essere l'inserimento di un metodo **clearSpitValve()** nella classe di base **Instrument**; questo, tuttavia, potrebbe generare confusione poiché implicherebbe che anche gli strumenti **Percussion**, **Stringed** ed **Electronic** abbiano valvole di sfiato. RTTI consente di implementare una soluzione più ottimale, grazie all'inserimento del metodo nella classe che ritenete appropriata, in questo caso gli strumenti **Wind**. Contemporaneamente, potreste rendervi conto che esiste un'altra soluzione, ancora più ragionevole: utilizzare un metodo **prepareInstrument()** nella classe di base. Tuttavia, visto che non avete notato questa possibilità quando avete iniziato a scrivere il programma, per ovviare a questo inconveniente dovete servirvi dell'RTTI.

In alcune occasioni RTTI risolverà anche problemi di efficienza. Supponete che il vostro codice applichi il polimorfismo in modo appropriato, ma che uno dei vostri oggetti reagisca a questo codice multiuso in modo terribilmente inefficiente. Potreste selezionare il tipo che dà problemi di efficienza e utilizzare RTTI per migliorare il codice specifico. Diffidate, in ogni caso, del miglioramento nelle prestazioni, poiché è un processo che non deve essere intrapreso troppo presto: è una sirena da cui molti si lasciano sedurre. Di norma è preferibile ottenere innanzitutto un primo programma funzionante, poi valutare se le sue prestazioni sono adeguate. Solo in quel momento dovreste affrontare il problema dell'efficienza, per esempio con un *profiler*: a questo proposito consultate il supplemento disponibile su *http://mindview.net/Books/BetterJava*.

Avete anche visto che la riflessione apre un nuovo mondo di possibilità di programmazione, permettendo di implementare uno stile molto più dinamico. Per alcuni la natura dinamica della riflessione risulta fastidiosa: a una mente impigrita dalla sicurezza dei controlli di tipo statico, il fatto di poter eseguire operazioni che possono essere controllate in fase di esecuzione e segnalate soltanto con eccezioni potrebbe sembrare il percorso sbagliato.

Alcuni arrivano a sostenere che introdurre la possibilità di un'eccezione in fase di esecuzione è un chiaro indicatore che il codice (che contiene l'eccezione) dovrebbe essere evitato. L'autore ritiene che questo senso di sicurezza sia illusorio. In fase di esecuzione può sempre accadere qualcosa che solleva eccezioni, anche in un programma privo di blocchi **try** o specifiche di eccezione; invece l'esistenza di un modello coerente di gestione degli errori autorizza a scrivere codice dinamico che utilizza la riflessione. Naturalmente, quando è possibile vale la pena scrivere codice controllabile in modo statico. Il codice dinamico è tuttavia una delle importanti caratteristiche di Java.

**Esercizio 26** (3) Implementate il metodo **clearSpitValve()** come descritto nel sommario.

*La soluzione degli esercizi è disponibile nel documento The Thinking in Java Annotated Solution Guide, in vendita all'indirizzo www.mindview.net.*

# Capitolo 3

# Generici

La filosofia di base di Java è che "il codice malfatto non sarà eseguito". Le classi e i metodi normali funzionano con tipi specifici, primitivi o tipi di classe. Se scrivete codice che potrebbe essere utilizzato con più tipi, questa rigidità può essere eccessiva.[1]

Una delle tecniche adottate dai linguaggi orientati agli oggetti per consentire la generalizzazione è il polimorfismo. Potete scrivere, per esempio, un metodo che accetta come argomento una classe di base, quindi utilizzare quel metodo con qualunque classe sia derivata da quella classe di base. Ora il vostro metodo è un po' più generico e può essere utilizzato in vari punti del codice. La stessa funzionalità è disponibile per le classi: in qualsiasi punto utilizziate un tipo specifico, un tipo di base fornisce maggiore flessibilità. Naturalmente, tutto può essere esteso, tranne una classe di tipo **final** o una con tutti i costruttori **private**, pertanto questa flessibilità è quasi sempre garantita in modo automatico.

A volte, il fatto di essere limitati a una sola gerarchia è una condizione eccessivamente restrittiva: se l'argomento di un metodo è un'interfaccia anziché una classe,

---

1. Le FAQ di Angelika Langer sui generici di Java (*www.angelikalanger.com/GenericsFAq/JavaGenericsFaq.html*) e altri suoi scritti, in collaborazione con Klaus Kreft, sono stati preziosissimi nella stesura di questo capitolo.

le limitazioni si allentano per includere qualsiasi oggetto implementi l'interfaccia, incluse le classi che non avete ancora creato. Questo dà al programmatore client l'opportunità di implementare un'interfaccia per conformarsi alla vostra classe o al vostro metodo: questo è il motivo per cui le interfacce permettono di attraversare le gerarchie di classe, purché abbiate la possibilità di creare una nuova classe per farlo.

Spesso però anche un'interfaccia è troppo restrittiva, poiché richiede che il vostro codice lavori con quell'interfaccia particolare. Potreste scrivere codice ancora più generico se riusciste ad affermare che il vostro codice funziona con "un tipo non specificato", anziché con un'interfaccia o una classe specifica.

Questo è appunto il concetto dei generici, uno dei cambiamenti più importanti in Java SE5: i generici implementano il concetto dei *tipi parametrizzati*, i quali permettono di creare componenti (di norma contenitori) facili da utilizzare con tipi multipli. Il termine *generico* significa "riferito o adeguato a grandi gruppi di classi". Lo scopo originale dei generici nei linguaggi di programmazione era quello di garantire al programmatore la massima espressività possibile nella scrittura di classi o metodi, allentando i vincoli sui tipi con cui funzionano questi oggetti. Come vedrete in questo capitolo, l'implementazione Java dei generici non ha una portata così ampia, al punto che potreste chiedervi se il termine *generici* sia davvero appropriato per questa funzionalità.

Se non avete mai visto in azione un qualunque genere di tipo parametrizzato, il meccanismo dei generici di Java probabilmente vi sembrerà una pratica integrazione al linguaggio. Quando create un'istanza di un tipo parametrizzato questo meccanismo si occupa di gestire automaticamente il casting e garantisce la correttezza dei tipi in fase di compilazione: un miglioramento, all'apparenza.

Tuttavia, se già conoscete il funzionamento dei tipi parametrizzati, per esempio in C++, troverete che i generici di Java non vi consentono l'ampio spazio di manovra che vi aspettereste. Benché l'utilizzo dei tipi generici di altri programmatori sia ragionevolmente semplice, quando si tratta di generare i vostri generici avrete numerose sorprese. Una delle cose che si cercherà di spiegare è come questa funzionalità sia potuta diventare ciò che è attualmente.

Con questo non si può affermare che i generici di Java non siano utili: in molti casi rendono il codice più diretto, e persino elegante. Tuttavia, se provenite da un linguaggio che implementa una versione più pura dei generici potreste rimanere delusi. In questo capitolo esaminerete sia i punti di forza sia le limitazioni che caratterizzano i generici di Java, in modo da poter utilizzare questa nuova funzionalità nel modo più efficace.

## Confronto con C++

I progettisti di Java hanno sostenuto che buona parte dell'ispirazione per questo linguaggio è venuta per reazione a C++. Malgrado ciò, è possibile insegnare Java quasi senza fare riferimento a C++: l'autore ha cercato di adottare questo approccio, tranne che nelle situazioni in cui il confronto tra i due linguaggi avrebbe garantito un maggiore livello di comprensione.

I generici richiedono un parallelo più ravvicinato con C++, per due motivi. In primo luogo, la comprensione di determinati aspetti dei *template* di C++ (la principale fonte di ispirazione per i generici, inclusa la sintassi di base) vi aiuterà non solo a capire i concetti fondamentali, ma soprattutto i limiti di quanto è possibile realizzare con i generici Java e le relative cause. L'obiettivo finale è farvi comprendere in che cosa consistono i limiti perché, secondo l'esperienza dell'autore, la conoscenza dei limiti di un linguaggio contribuisce a formare i programmatori migliori. Essendo consapevoli di ciò che non vi è possibile ottenere, riuscirete a fare un utilizzo migliore di ciò che invece potete fare, in parte anche perché eviterete di perdere tempo rimbalzando contro muri di gomma.

Il secondo motivo è che nella comunità Java vi è un diffuso malinteso a proposito dei template di C++, il quale potrebbe ulteriormente confondervi riguardo agli scopi dei generici.

Pertanto, nonostante in questo capitolo l'autore abbia scelto di illustrarvi alcuni esempi dei template di C++, ha deciso di non eccedere.

## Generici semplici

Una delle motivazioni iniziali per l'utilizzo dei generici è generare le classi contenitore, un argomento che avete visto nel Volume 1, Capitolo 11 e che avrete modo di approfondire ulteriormente nel Capitolo 5. Un contenitore è un "oggetto" in cui conservare gli oggetti con i quali state lavorando. Sebbene la stessa caratteristica sia offerta dagli array, i contenitori si dimostrano più flessibili e offrono funzionalità diverse dai comuni array. Praticamente qualsiasi programma richiede che manteniate un gruppo degli oggetti mentre li utilizzate, quindi i contenitori rappresentano una delle librerie di classe maggiormente riutilizzabili.

Considerate una classe che contiene un solo oggetto; ovviamente la classe potrebbe specificare il tipo esatto dell'oggetto, come in questo esempio:

```
//: generics/Holder1.java

class Automobile {}
```

```
  public class Holder1 {
   private Automobile a;
   public Holder1(Automobile a) { this.a = a; }
   Automobile get() { return a; }
 } ///:~
```

Tuttavia questo non è uno "strumento" particolarmente riutilizzabile, perché non può essere sfruttato per contenere nient'altro: sarebbe preferibile non doverlo riscrivere di nuovo per ogni tipo che si incontra.

Prima di Java SE5, avreste semplicemente fatto in modo di fargli contenere un **Object**:

```
//: generics/Holder2.java

public class Holder2 {
  private Object a;
public Holder2(Object a) { this.a = a; }
  public void set(Object a) { this.a = a; }
  public Object get() { return a; }
  public static void main(String[] args) {
    Holder2 h2 = new Holder2(new Automobile());
    Automobile a = (Automobile)h2.get();
    h2.set("Not an Automobile");
    String s = (String)h2.get();
    h2.set(1); // Esegue l'autoboxing a Integer
    Integer x = (Integer)h2.get();
  }
} ///:~
```

Ora un **Holder2** può contenere qualsiasi oggetto, e in questo esempio specifico ne contiene di tre tipi differenti.

In alcuni casi si richiede che un contenitore possa adattarsi a molteplici tipi di oggetti, ma in genere il contenitore è utilizzato per contenere un solo tipo di oggetto. Una delle motivazioni primarie per i generici è specificare quale tipo di oggetto dovrà conservare un contenitore, e fare in modo che questa specifica sia supportata dal compilatore.

Quindi, invece di **Object** sarebbe preferibile utilizzare un tipo non specificato, che potrà essere scelto in seguito. A questo scopo, inserite un *tipo parametrizzato* tra

i simboli di minore e di maggiore (< e >) dopo il nome della classe, che sostituirete con il tipo effettivo al momento di utilizzare la classe. Per la classe "holder" il risultato sarà simile al seguente, in cui **T** è il parametro che indica il tipo:

```
//: generics/Holder3.java

public class Holder3<T> {
  private T a;
  public Holder3(T a) { this.a = a; }
  public void set(T a) { this.a = a; }
  public T get() { return a; }
  public static void main(String[] args) {
    Holder3<Automobile> h3 =
      new Holder3<Automobile>(new Automobile());
    Automobile a = h3.get(); // Il casting non e' necessario
    // h3.set("Not an Automobile"); // Errore
    // h3.set(1); // Errore
  }
} ///:~
```

Quando create un **Holder3** dovete specificare il tipo che desiderate mettere in opera, servendovi della stessa sintassi con i simboli < e >, come potete vedere in **main()**. In **Holder3** avete la possibilità di inserire soltanto oggetti di quel tipo o di un sottotipo, dal momento che il principio della sostituzione si applica anche con i generici: il valore che estrarrete sarà automaticamente del tipo corretto.

Questa è l'idea di base dei generici di Java: indicate il tipo che volete utilizzare, e la funzionalità si prende cura dei dettagli.

In generale, potete gestire i generici come se si trattasse di un qualsiasi altro tipo: il fatto che abbiano parametri di tipo è ininfluente; tuttavia, come vedrete, potete utilizzare i generici semplicemente richiamandoli con il loro elenco di argomenti di tipo.

> **Esercizio 1** (1) Utilizzate **Holder3** con la libreria **typeinfo.pets**, per indicare che un **Holder3**, il quale è previsto contenere un tipo di base, può contenere anche un tipo derivato.
>
> **Esercizio 2** (1) Generate una classe "contenitore" che contenga tre oggetti dello stesso tipo, con i metodi per registrare ed estrarre questi tre oggetti e un costruttore che li inizializzi.

## *Una libreria di tuple*

Una delle necessità più ricorrenti in Java è fare in modo che una chiamata di metodo restituisca oggetti multipli. La dichiarazione di ritorno vi consente di specificare un solo oggetto, quindi la soluzione consiste nel generare un oggetto il quale contenga gli oggetti multipli che volete restituire. Ovviamente potreste scrivere un classe speciale ogni volta che avete questo problema, ma i generici vi permettono di risolverlo definitivamente e nello stesso tempo vi garantiscono la sicurezza dei tipi (*type-safety*) in fase di compilazione.

Questo concetto è denominato *tupla*, "oggetto di trasferimento dati" (*Data Transfer Object*) o *messenger*, ed è rappresentato semplicemente da un gruppo di oggetti inglobati in un unico oggetto. Il destinatario dell'oggetto può leggere gli elementi, ma non inserirne di nuovi.

Di norma, le tuple possono essere di qualsiasi dimensione e ogni oggetto contenuto può essere di un tipo diverso. Tuttavia, vi aspettereste di poter specificare il tipo di ogni oggetto, nonché di potervi accertare che quando il destinatario otterrà il valore, questo sia del tipo corretto. Per gestire le lunghezze multiple, dovrete generare diverse tuple. Eccone una che contiene due oggetti:

```
//: net/mindview/util/TwoTuple.java
package net.mindview.util;

public class TwoTuple<A,B> {
  public final A first;
  public final B second;
  public TwoTuple(A a, B b) { first = a; second = b; }
  public String toString() {
    return "(" + first + ", " + second + ")";
  }
} ///:~
```

Il costruttore intercetta l'oggetto da immagazzinare, mentre **toString()** è una funzione di comodo per visualizzare i valori in un elenco; notate che la tupla mantiene implicitamente ordinati gli elementi.

A una prima lettura, potreste pensare che questo codice violi i principi di sicurezza comuni della programmazione Java: infatti, **first** e **second** non dovrebbero essere **private** e accessibili soltanto tramite i metodi **getFirst()** e **getSecond()**? Considerate la sicurezza che otterreste in tal caso: i programmatori client potrebbero leggere senza problema gli oggetti e farne l'utilizzo

che desiderano, ma non potrebbero assegnare **first** e **second** a nient'altro. La dichiarazione **final** vi garantisce la stessa sicurezza, ma in una forma più concisa e semplice.

Un'altra osservazione progettuale è che potreste *volere* permettere che un programmatore client faccia puntare **first** o **second** a un altro oggetto. Risulta tuttavia più sicuro lasciare **first** e **second** nella forma in cui si trovano e costringere il programmatore utente a generare una nuova **TwoTuple**, se desidera disporre di elementi diversi.

Tramite l'ereditarietà, potete creare tuple di lunghezza superiore; come mostra questo esempio, l'aggiunta di altri parametri di tipo è molto semplice:

```
//: net/mindview/util/ThreeTuple.java
package net.mindview.util;

public class ThreeTuple<A,B,C> extends TwoTuple<A,B> {
  public final C third;
  public ThreeTuple(A a, B b, C c) {
    super(a, b);
    third = c;
  }
  public String toString() {
    return "(" + first + ", " + second + ", " + third +")";
  }
} ///:~
//: net/mindview/util/FourTuple.java
package net.mindview.util;

public class FourTuple<A,B,C,D> extends ThreeTuple<A,B,C> {
  public final D fourth;
  public FourTuple(A a, B b, C c, D d) {
    super(a, b, c);
    fourth = d;
  }
  public String toString() {
    return "(" + first + ", " + second + ", " +
      third + ", " + fourth + ")";
  }
```

```
} ///:~

//: net/mindview/util/FiveTuple.java
package net.mindview.util;

public class FiveTuple<A,B,C,D,E>
extends FourTuple<A,B,C,D> {
  public final E fifth;
  public FiveTuple(A a, B b, C c, D d, E e) {
    super(a, b, c, d);
    fifth = e;
  }
  public String toString() {
    return "(" + first + ", " + second + ", " +
      third + ", " + fourth + ", " + fifth + ")";
  }
} ///:~
```

Per utilizzare una tupla, definite semplicemente quella di lunghezza opportuna come valore di ritorno della vostra funzione, quindi create e restituite la tupla nella dichiarazione **return**:

```
//: generics/TupleTest.java
import net.mindview.util.*;

class Amphibian {}
class Vehicle {}

public class TupleTest {
  static TwoTuple<String,Integer> f() {
    // L'autoboxing converte int in Integer:
    return new TwoTuple<String,Integer>("hi", 47);
  }
  static ThreeTuple<Amphibian,String,Integer> g() {
    return new ThreeTuple<Amphibian, String, Integer>(
      new Amphibian(), "hi", 47);
  }
```

```
  static
  FourTuple<Vehicle,Amphibian,String,Integer> h() {
    return
      new FourTuple<Vehicle,Amphibian,String,Integer>(
        new Vehicle(), new Amphibian(), "hi", 47);
  }
  static
  FiveTuple<Vehicle,Amphibian,String,Integer,Double> k() {
    return new
      FiveTuple<Vehicle,Amphibian,String,Integer,Double>(
        new Vehicle(), new Amphibian(), "hi", 47, 11.1);
  }
  public static void main(String[] args) {
    TwoTuple<String,Integer> ttsi = f();
    System.out.println(ttsi);
    // ttsi.first = "there"; // Errore di compilazione: final
    System.out.println(g());
    System.out.println(h());
    System.out.println(k());
  }
} /* Output: (80% match)
(hi, 47)
(Amphibian@1f6a7b9, hi, 47)
(Vehicle@35ce36, Amphibian@757aef, hi, 47)
(Vehicle@9cab16, Amphibian@1a46e30, hi, 47, 11.1)
*///:~
```

Grazie ai generici, potete creare senza problemi qualsiasi tupla che restituisca qualunque gruppo di tipi semplicemente scrivendo l'espressione.

Il fallimento della dichiarazione **ttsi.first = "there";** dimostra come la specifica **final** sui campi pubblici ne impedisca la riassegnazione dopo la costruzione.

Le espressioni **new** sono più verbose, e nel prosieguo del capitolo vedrete come sia possibile semplificarle ricorrendo ai *metodi generici*.

**Esercizio 3** (1) Create e testate una classe **SixTuple** generica.

**Esercizio 4** (3) Rendete generico il codice di **innerclasses/Sequence.java**.

## *Una classe di stack*

Ora affronterete un argomento leggermente più complesso: il tradizionale stack di tipo *pushdown*. Nel Volume 1, Capitolo 11 è stato illustrato come implementare questo stack ricorrendo a una **LinkedList** come classe **net. mindview.util.Stack**; in quell'esempio avete visto che una **LinkedList** dispone già dei metodi necessari per creare uno stack. Lo stack è stato realizzato componendo una classe generica (**Stack<T>**) con un'altra classe generica (**LinkedList<T>**); avrete certamente notato che, fatte salve alcune eccezioni che vedrete in seguito, un tipo generico non è altro che un tipo qualsiasi.

Invece di utilizzare **LinkedList**, potete così implementare una versione personalizzata del meccanismo di archiviazione collegato internamente.

```
//: generics/LinkedStack.java
// Uno stack implementato con una struttura collegata
// internamente.

public class LinkedStack<T> {
  private static class Node<U> {
    U item;
    Node<U> next;
    Node() { item = null; next = null; }
    Node(U item, Node<U> next) {
      this.item = item;
      this.next = next;
    }
    boolean end() { return item == null && next == null; }
  }
  private Node<T> top = new Node<T>(); // Fine sentinella
  public void push(T item) {
    top = new Node<T>(item, top);
  }
  public T pop() {
    T result = top.item;
    if(!top.end())
      top = top.next;
    return result;
  }
```

```
    public static void main(String[] args) {
      LinkedStack<String> lss = new LinkedStack<String>();
      for(String s : "Phasers on stun!".split(" "))
        lss.push(s);
      String s;
      while((s = lss.pop()) != null)
        System.out.println(s);
    }
  } /* Output:
  stun!
  on
  Phasers
  *///:~
```

Anche la classe interna **Node** è un generico con il suo parametro di tipo.

In questo esempio, per determinare quando lo stack è vuoto si utilizza un "valore sentinella", che l'autore chiama *end sentinel* (fine sentinella).

Il fine sentinella viene creato al momento di costruire il **LinkedStack**, e ogni volta che chiamate **push()** viene generato un nuovo **Node<T>** poi collegato al **Node<T>** precedente. Quando chiamate **pop()**, restituite sempre il **top.item**: a quel punto scartate il **Node<T>** corrente e vi spostate su quello successivo; quando incontrate il fine sentinella, invece, rimanete dove siete. In questo modo, se il client continuerà a chiamare **pop()** otterrà **null**, a dimostrazione che lo stack è vuoto.

**Esercizio 5** (2) Rimuovete il parametro di tipo dalla classe **Node** e modificate il resto del codice in **LinkedStack.java**, per illustrare che una classe interna ha accesso ai parametri di tipo generico della sua classe esterna.

## RandomList

Come ulteriore esempio di "holder", immaginate di avere bisogno di un tipo speciale di elenco che selezioni in modo casuale uno dei suoi elementi ogni volta che chiamate **select()**. Volete approfittare della realizzazione di questo codice per costruire uno strumento che funzioni con tutti gli oggetti, pertanto decidete di servirvi dei generici:

```
//: generics/RandomList.java
import java.util.*;
```

```
public class RandomList<T> {
  private ArrayList<T> storage = new ArrayList<T>();
  private Random rand = new Random(47);
  public void add(T item) { storage.add(item); }
  public T select() {
    return storage.get(rand.nextInt(storage.size()));
  }
  public static void main(String[] args) {
    RandomList<String> rs = new RandomList<String>();
    for(String s: ("The quick brown fox jumped over " +
        "the lazy brown dog").split(" "))
      rs.add(s);
    for(int i = 0; i < 11; i++)
      System.out.print(rs.select() + " ");
  }
} /* Output:
brown over fox quick quick dog brown the brown lazy brown
*///:~
```

**Esercizio 6** (1) Utilizzate **RandomList** con due nuovi tipi in aggiunta a quello presente in **main()**.

## Interfacce generiche

I generici funzionano anche con le interfacce; per esempio, un *generatore* è una classe che crea oggetti. In realtà si tratta di una specializzazione del *design pattern Factory Method*, ma quando chiedete a un generatore un nuovo oggetto non passate alcun argomento, diversamente da quanto fareste con il Factory Method: il generatore sa infatti come produrre nuovi oggetti senza bisogno di alcuna informazione.

Tipicamente un generatore definisce un solo metodo, quello che produce i nuovi oggetti; in questo caso, lo chiamerete **next()** e lo includerete nei programmi di utilità standard:

```
//: net/mindview/util/Generator.java
// Un'interfaccia generica.
package net.mindview.util;
public interface Generator<T> { T next(); } ///:~
```

Il tipo di ritorno di **next()** è parametrizzato a **T**. Come vedete, l'impiego dei generici con le interfacce non è diverso dall'utilizzo dei generici con le classi.

Per dimostrare l'implementazione di un **Generator** vi occorrono alcune classi. A questo proposito, ecco una gerarchia di caffè:

```
//: generics/coffee/Coffee.java
package generics.coffee;

public class Coffee {
  private static long counter = 0;
  private final long id = counter++;
  public String toString() {
    return getClass().getSimpleName() + " " + id;
  }
} ///:~

//: generics/coffee/Latte.java
package generics.coffee;
public class Latte extends Coffee {} ///:~

//: generics/coffee/Mocha.java
package generics.coffee;
public class Mocha extends Coffee {} ///:~

//: generics/coffee/Cappuccino.java
package generics.coffee;
public class Cappuccino extends Coffee {} ///:~

//: generics/coffee/Americano.java
package generics.coffee;
public class Americano extends Coffee {} ///:~

//: generics/coffee/Breve.java
package generics.coffee;
public class Breve extends Coffee {} ///:~
```

Ora potete implementare un **Generator<Coffee>** che produca in modo casuale i diversi tipi di oggetti **Coffee**:

```
//: generics/coffee/CoffeeGenerator.java
// Genera diversi tipi di caffè (Coffee):
package generics.coffee;
import java.util.*;
import net.mindview.util.*;

public class CoffeeGenerator
implements Generator<Coffee>, Iterable<Coffee> {
  private Class[] types = { Latte.class, Mocha.class,
    Cappuccino.class, Americano.class, Breve.class, };
  private static Random rand = new Random(47);
  public CoffeeGenerator() {}
  // Per l'iterazione:
  private int size = 0;
  public CoffeeGenerator(int sz) { size = sz; }
  public Coffee next() {
    try {
      return (Coffee)
        types[rand.nextInt(types.length)].newInstance();
      // Segnala gli errori in fase di esecuzione:
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
  class CoffeeIterator implements Iterator<Coffee> {
    int count = size;
    public boolean hasNext() { return count > 0; }
    public Coffee next() {
      count--;
      return CoffeeGenerator.this.next();
    }
    public void remove() { // Non implementato
      throw new UnsupportedOperationException();
```

```
    }
  };
  public Iterator<Coffee> iterator() {
    return new CoffeeIterator();
  }
  public static void main(String[] args) {
    CoffeeGenerator gen = new CoffeeGenerator();
    for(int i = 0; i < 5; i++)
      System.out.println(gen.next());
    for(Coffee c : new CoffeeGenerator(5))
      System.out.println(c);
  }
} /* Output:
Americano 0
Latte 1
Americano 2
Mocha 3
Mocha 4
Breve 5
Americano 6
Latte 7
Cappuccino 8
Cappuccino 9
*///:~
```

L'interfaccia parametrizzata di **Generator** si assicura che **next()** restituisca il tipo di parametro. **CoffeeGenerator** implementa anche l'interfaccia **Iterable**, pertanto è utilizzabile in una dichiarazione foreach; richiede tuttavia un "fine sentinella" per sapere quando arrestarsi, che viene prodotto dal secondo costruttore.

Ecco una seconda implementazione di **Generator<T>**, questa volta per produrre i numeri di Fibonacci:

```
//: generics/Fibonacci.java
// Genera una sequenza di Fibonacci.
import net.mindview.util.*;

public class Fibonacci implements Generator<Integer> {
```

```
  private int count = 0;
  public Integer next() { return fib(count++); }
  private int fib(int n) {
    if(n < 2) return 1;
    return fib(n-2) + fib(n-1);
  }
  public static void main(String[] args) {
    Fibonacci gen = new Fibonacci();
    for(int i = 0; i < 18; i++)
      System.out.print(gen.next() + " ");
  }
} /* Output:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
*///:~
```

Anche se state lavorando con **int** sia all'interno sia all'esterno della classe, il parametro di tipo è **Integer**. Questo dettaglio evidenzia una delle limitazioni dei generici Java: non permettono di utilizzare i primitivi come parametri di tipo. Fortunatamente, Java SE5 ha integrato la funzionalità di autoboxing e di "autounboxing" per convertire da tipi primitivi a tipi wrapper, e viceversa: potete vedere l'effetto di queste funzionalità perché gli **int** vengono utilizzati e prodotti dalla classe senza alcun problema.

Ma se volete andare più a fondo, perché non creare un generatore di numeri di Fibonacci di tipo **Iterable**? Una tecnica consiste nell'implementare nuovamente la classe aggiungendovi l'interfaccia **Iterable**, ma non sempre si dispone del codice originale e riscriverlo partendo da zero sarebbe un inutile spreco di risorse. Potete invece generare un *adattatore* per produrre l'interfaccia voluta, ricorrendo al design pattern Adapter già illustrato.

Gli adattatori possono essere implementati in vari modi, per esempio mediante l'ereditarietà, che genererà la classe adattata:

```
//: generics/IterableFibonacci.java
// Adatta la classe Fibonacci per renderla Iterable.
import java.util.*;

public class IterableFibonacci
extends Fibonacci implements Iterable<Integer> {
  private int n;
```

```
    public IterableFibonacci(int count) { n = count; }
    public Iterator<Integer> iterator() {
      return new Iterator<Integer>() {
        public boolean hasNext() { return n > 0; }
        public Integer next() {
          n--;
          return IterableFibonacci.this.next();
        }
        public void remove() { // Non implementato
          throw new UnsupportedOperationException();
        }
      };
    }
    public static void main(String[] args) {
      for(int i : new IterableFibonacci(18))
        System.out.print(i + " ");
    }
  } /* Output:
  1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
  *///:~
```

Per impiegare **IterableFibonacci** in una dichiarazione foreach dovete assegnare dei limiti al costruttore, in modo che **hasNext()** sappia quando restituire **false**.

**Esercizio 7** (2) Mediante composizione e non ereditarietà, adattate **Fibonacci.java** per renderlo **Iterable**.

**Esercizio 8** (2) Seguendo lo schema dell'esempio **Coffee**, generate una gerarchia di **StoryCharacters** del vostro film preferito dividendo i personaggi in buoni e cattivi, **GoodGuys** e **BadGuys**. Create un generatore per **StoryCharacters** sul modello di **CoffeeGenerator**.

## Metodi generici

Finora avete visto come parametrizzare intere classi, ma ricorderete che è possibile farlo anche con i metodi all'interno di una classe. La classe in sé può essere o non essere generica: questa è una condizione indipendente dall'avere un metodo generico.

Rendere un metodo generico significa permettergli di variare a prescindere dalla classe. Di norma dovreste utilizzare i metodi generici ogni volta che vi sia possibile: in pratica, se avete l'opportunità di rendere generico un metodo invece che l'intera classe, è probabilmente meglio che lavoriate sul primo. Inoltre, se il metodo è **static** non ha accesso ai parametri di tipo generico della classe, pertanto deve essere reso generalizzato, dovrà essere un metodo generico.

Per definire un metodo generico, basta indicare un elenco di parametri generici prima del valore di ritorno, come in questo esempio:

```
//: generics/GenericMethods.java

public class GenericMethods {
  public <T> void f(T x) {
    System.out.println(x.getClass().getName());
  }
  public static void main(String[] args) {
    GenericMethods gm = new GenericMethods();
    gm.f("");
    gm.f(1);
    gm.f(1.0);
    gm.f(1.0F);
    gm.f('c');
    gm.f(gm);
  }
} /* Output:
java.lang.String
java.lang.Integer
java.lang.Double
java.lang.Float
java.lang.Character
GenericMethods
*///:~
```

La classe **GenericMethods** non è parametrizzata, anche se una classe e i suoi metodi possono entrambi esserlo nello stesso tempo: in questo caso, però, soltanto il metodo **f()** ha un parametro di tipo, indicato dall'elenco di parametri che precede il tipo di ritorno del metodo.

Ricordate che per una classe generica occorre specificare i parametri di tipo al momento di istanziare la classe. Con un metodo generico, tuttavia, di solito non occorre specificare i tipi di parametro poiché il compilatore è in grado di determinarli autonomamente, secondo una tecnica denominata *deduzione del tipo* o *inferenza degli argomenti di tipo* (*type argument inference*). Per questo motivo, le chiamate a **f()** ricordano le normali chiamate di metodo e il metodo sembra **f()** sovraccaricato all'infinito, arrivando ad accettare persino un argomento del tipo **GenericMethods**.

Per le chiamate a **f()** che utilizzano tipi primitivi entra in gioco l'autoboxing, che automaticamente ingloba i tipi primitivi negli oggetti loro collegati. In effetti, i metodi generici e l'autoboxing possono eliminare una parte di codice che in precedenza sarebbe stata necessaria per la conversione manuale.

**Esercizio 9** (1) Modificate **GenericMethods.java** in modo che **f()** accetti tre argomenti, tutti di tipi parametrizzati differenti.

**Esercizio 10** (1) Modificare l'Esercizio 9 in modo che uno degli argomenti di **f()** non sia parametrizzato.

## *Sfruttare la deduzione del tipo*

Tra le rimostranze che vengono segnalate a proposito dei generici, vi è il fatto che aumentano la quantità di codice richiesto. Considerate, per esempio, il programma **holding/MapOfList.java** del Volume 1, Capitolo 11, nel quale la creazione della **Map** di **List** richiede il codice seguente:

```
public static Map<Person, List<? extends Pet>>
    petPeople = new HashMap<Person, List<? extends Pet>>();
```

Ignorate per il momento l'utilizzo di **extends** e del punto interrogativo, che saranno illustrati nel prosieguo del capitolo; per il resto, sembra quasi che vi stiate ripetendo e che il compilatore debba determinare ciascuna delle liste di argomenti generici dall'altra. Purtroppo non può, ma la tecnica di deduzione in un metodo generico può garantire una certa semplificazione. Per esempio, potreste creare un'utility contenente vari metodi **static** che produca le implementazioni più comunemente utilizzate per i diversi contenitori:

```
//: net/mindview/util/New.java
// Utilita' per semplificare la creazione di contenitori
// generici ricorrendo alla deduzione degli argomenti di tipo.
package net.mindview.util;
import java.util.*;
```

```
public class New {
  public static <K,V> Map<K,V> map() {
    return new HashMap<K,V>();
  }
  public static <T> List<T> list() {
    return new ArrayList<T>();
  }
  public static <T> LinkedList<T> lList() {
    return new LinkedList<T>();
  }
  public static <T> Set<T> set() {
    return new HashSet<T>();
  }
  public static <T> Queue<T> queue() {
    return new LinkedList<T>();
  }
  // Esempi:
  public static void main(String[] args) {
    Map<String, List<String>> sls = New.map();
    List<String> ls = New.list();
    LinkedList<String> lls = New.lList();
    Set<String> ss = New.set();
    Queue<String> qs = New.queue();
  }
} ///:~
```

In **main()**, potete notare alcuni esempi di impiego della deduzione per ovviare alla necessità di ripetere l'elenco di parametri generici. La deduzione del tipo può essere applicata a **holding/MapOfList.java**:

```
//: generics/SimplerPets.java
import typeinfo.pets.*;
import java.util.*;
import net.mindview.util.*;

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

```
    Map<Person, List<? extends Pet>> petPeople = New.map();
    // Il resto del codice e' identico...
  }
} ///:~
```

Benché questo sia un esempio interessante della deduzione del tipo, è difficile dire quanto questa tecnica sia realmente convincente. La persona che legge il codice è tenuta ad analizzare e capire questa ulteriore libreria e le relative implicazioni, di conseguenza potrebbe essere altrettanto produttivo lasciare la definizione originale, per quanto ripetitiva (ironicamente, per semplicità). Tuttavia, se alla libreria Java standard venisse integrata una funzionalità analoga al programma di utilità **New.java** precedente, avrebbe senso servirsi della deduzione.

La deduzione del tipo funziona soltanto per le assegnazioni: se passate il risultato di una chiamata di metodo quale **New.map()** come argomento a un altro metodo, il compilatore non cercherà di eseguire la deduzione, ma considererà la chiamata di metodo come se il valore di ritorno fosse assegnato a una variabile di tipo **Object**. Ecco un esempio che non funzionerà:

```
//: generics/LimitsOfInference.java
import typeinfo.pets.*;
import java.util.*;

public class LimitsOfInference {
  static void
  f(Map<Person, List<? extends Pet>> petPeople) {}
  public static void main(String[] args) {
    // f(New.map()); // Non compila
  }
} ///:~
```

**Esercizio 11** (1) Testate **New.java** generando le vostre classi e assicurandovi che funzionino correttamente con **New**.

## Specifica di tipo esplicita

È possibile specificare in modo esplicito il tipo in un metodo generico, sebbene questa forma di sintassi sia raramente necessaria. A tale scopo, inserite il tipo tra i simboli < e > dopo il punto (.) e immediatamente prima del nome del metodo. Quando chiamate un metodo dall'interno della

stessa classe dovrete specificare **this** prima del punto, e quando lavorate con i metodi **static** dovrete indicare il nome della classe prima del punto. Il problema esposto in **LimitsOfInference.java** può essere risolto utilizzando la sintassi seguente:

```
//: generics/ExplicitTypeSpecification.java
import typeinfo.pets.*;
import java.util.*;
import net.mindview.util.*;

public class ExplicitTypeSpecification {
  static void f(Map<Person, List<Pet>> petPeople) {}
  public static void main(String[] args) {
    f(New.<Person, List<Pet>>map());
  }
} ///:~
```

Naturalmente questa forma elimina il beneficio di utilizzare la classe **New** per ridurre la quantità di codice da scrivere, ma la sintassi supplementare è richiesta soltanto se non state scrivendo una dichiarazione di assegnazione.

**Esercizio 12** (1) Ripetete l'esercizio precedente utilizzando la specifica di tipo esplicita.

## *Elenchi di argomenti variabili e metodi generici*

I metodi generici e gli elenchi di argomenti possono coesistere senza problemi:

```
//: generics/GenericVarargs.java
import java.util.*;

public class GenericVarargs {
  public static <T> List<T> makeList(T... args) {
    List<T> result = new ArrayList<T>();
    for(T item : args)
      result.add(item);
    return result;
```

```
  }
  public static void main(String[] args) {
    List<String> ls = makeList("A");
    System.out.println(ls);
    ls = makeList("A", "B", "C");
    System.out.println(ls);
    ls = makeList("ABCDEFFHIJKLMNOPQRSTUVWXYZ".split(""));
    System.out.println(ls);
  }
} /* Output:
[A]
[A, B, C]
[, A, B, C, D, E, F, F, H, I, J, K, L, M, N, O, P, Q, R, S,
T, U, V, W, X, Y, Z]
*///:~
```

Il metodo **makeList()** di questo esempio offre la stessa funzionalità del metodo **java.util.Arrays.asList()** della libreria standard.

## *Un metodo generico da utilizzare con i Generator*

È appropriato utilizzare un generatore per riempire una **Collection** e ha perfettamente senso "generalizzare" questa operazione:

```
//: generics/Generators.java
// Una utility da usare con i Generator.
import generics.coffee.*;
import java.util.*;
import net.mindview.util.*;

public class Generators {
  public static <T> Collection<T>
  fill(Collection<T> coll, Generator<T> gen, int n) {
    for(int i = 0; i < n; i++)
      coll.add(gen.next());
    return coll;
  }
```

```
  public static void main(String[] args) {
    Collection<Coffee> coffee = fill(
      new ArrayList<Coffee>(), new CoffeeGenerator(), 4);
    for(Coffee c : coffee)
      System.out.println(c);
    Collection<Integer> fnumbers = fill(
      new ArrayList<Integer>(), new Fibonacci(), 12);
    for(int i : fnumbers)
      System.out.print(i + ", ");
  }
} /* Output:
Americano 0
Latte 1
Americano 2
Mocha 3
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
*///:~
```

Osservate come il metodo generico **fill()** sia applicato in modo trasparente ai contenitori e ai generatori **Coffee** e **Integer**.

**Esercizio 13** (4) Sovraccaricate il metodo **fill()** in modo che gli argomenti e i tipi di ritorno siano i sottotipi specifici di **Collection**: **List**, **Queue** e **Set**. In questo modo non perderete il tipo di contenitore. Siete in grado di eseguire l'overload per distinguere tra **List** e **LinkedList**?

## *Un Generator multiutilizzo*

Questa è una classe che produce un **Generator** per qualunque classe abbia un costruttore predefinito; per ridurre la quantità di codice da scrivere, la classe include un metodo generico per creare un **BasicGenerator**:

```
//: net/mindview/util/BasicGenerator.java
// Crea automaticamente un Generator, sulla base di una classe
// con un costruttore predefinito (senza argomenti).
package net.mindview.util;

public class BasicGenerator<T> implements Generator<T> {
```

```
    private Class<T> type;
    public BasicGenerator(Class<T> type){ this.type = type; }
    public T next() {
      try {
        // Da' per scontato che type sia una classe public:
        return type.newInstance();
      } catch(Exception e) {
        throw new RuntimeException(e);
      }
    }
    // Produce un generatore predefinito, sulla base di un
    // "contrassegno" di tipo (token):
    public static <T> Generator<T> create(Class<T> type) {
      return new BasicGenerator<T>(type);
    }
  } ///:~
```

Questa classe fornisce un'implementazione di base, la quale creerà oggetti di una classe che sia:

1. **public**, perché **BasicGenerator** è in un package separato e la classe in questione deve quindi avere diritti di accesso **public**, non soltanto quelli di pacchetto;
2. dotata di costruttore predefinito, ossia privo di argomenti.

Per generare un oggetto **BasicGenerator**, chiamate il metodo **create()** e passategli il contrassegno (*token*) per il tipo che desiderate venga prodotto. Il metodo generico **create()** consente di scrivere **BasicGenerator.create(MyType.class)** anziché **new BasicGenerator<MyType>(MyType.class)**, meno pratico ed elegante.

Per esempio, ecco una semplice classe con un costruttore predefinito:

```
//: generics/CountedObject.java

public class CountedObject {
  private static long counter = 0;
  private final long id = counter++;
  public long id() { return id; }
  public String toString() { return "CountedObject " + id;}
} ///:~
```

La classe **CountedObject** tiene traccia del numero di istanze di se stessa che sono state create e le segnala nel suo metodo **toString()**.

Servendovi di **BasicGenerator**, potete facilmente produrre un **Generator** per **CountedObject**:

```
//: generics/BasicGeneratorDemo.java
import net.mindview.util.*;

public class BasicGeneratorDemo {
  public static void main(String[] args) {
    Generator<CountedObject> gen =
      BasicGenerator.create(CountedObject.class);
for(int i = 0; i < 5; i++)
      System.out.println(gen.next());
  }
} /* Output:
CountedObject 0
CountedObject 1
CountedObject 2
CountedObject 3
CountedObject 4
*///:~
```

Potete vedere come il metodo generico riduca la quantità di codice necessario per creare l'oggetto **Generator**. I generici di Java vi costringono a passare comunque l'oggetto **Class**, che pertanto potreste utilizzare per la deduzione del tipo nel metodo **create()**.

> **Esercizio 14** (1) Modificate **BasicGeneratorDemo.java** affinché utilizzi la forma di creazione esplicita per il **Generator**, vale a dire il costruttore esplicito invece del metodo generico **create()**.

## *Semplificare l'utilizzo delle tuple*

La deduzione del tipo, insieme con le dichiarazioni di importazione **static**, consente di riscrivere le tuple viste in precedenza sotto forma di libreria a utilizzo generico. In questo caso, le tuple possono essere create ricorrendo a un metodo **static** sovraccarico:

```
//: net/mindview/util/Tuple.java
// Libreria di tuple che utilizza la deduzione del tipo.
```

```
package net.mindview.util;

public class Tuple {
  public static <A,B> TwoTuple<A,B> tuple(A a, B b) {
    return new TwoTuple<A,B>(a, b);
  }
  public static <A,B,C> ThreeTuple<A,B,C>
  tuple(A a, B b, C c) {
    return new ThreeTuple<A,B,C>(a, b, c);
  }
  public static <A,B,C,D> FourTuple<A,B,C,D>
  tuple(A a, B b, C c, D d) {
    return new FourTuple<A,B,C,D>(a, b, c, d);
  }
  public static <A,B,C,D,E>
  FiveTuple<A,B,C,D,E> tuple(A a, B b, C c, D d, E e) {
    return new FiveTuple<A,B,C,D,E>(a, b, c, d, e);
  }
} ///:~
```

Questa è una variante di **TupleTest.java** destinata a testare **Tuple.java**:

```
//: generics/TupleTest2.java
import net.mindview.util.*;
import static net.mindview.util.Tuple.*;

public class TupleTest2 {
  static TwoTuple<String,Integer> f() {
    return tuple("hi", 47);
  }
  static TwoTuple f2() { return tuple("hi", 47); }
  static ThreeTuple<Amphibian,String,Integer> g() {
    return tuple(new Amphibian(), "hi", 47);
  }
  static
  FourTuple<Vehicle,Amphibian,String,Integer> h() {
    return tuple(new Vehicle(), new Amphibian(), "hi", 47);
```

```
    }
    static
    FiveTuple<Vehicle,Amphibian,String,Integer,Double> k() {
      return tuple(new Vehicle(), new Amphibian(),
        "hi", 47, 11.1);
    }
    public static void main(String[] args) {
      TwoTuple<String,Integer> ttsi = f();
      System.out.println(ttsi);
      System.out.println(f2());
      System.out.println(g());
      System.out.println(h());
      System.out.println(k());
    }
  } /* Output: (80% match)
  (hi, 47)
  (hi, 47)
  (Amphibian@7ace8d, hi, 47)
  (Vehicle@ca2dce, Amphibian@8558d2, hi, 47)
  (Vehicle@a05308, Amphibian@ab50cd, hi, 47, 11.1)
  *///:~
```

Notate che **f()** restituisce un oggetto **TwoTuple** parametrizzato, mentre **f2()** restituisce un oggetto **TwoTuple** non parametrizzato. In questo caso il compilatore non produce alcun avvertimento relativo a **f2()**, perché il valore di ritorno non viene utilizzato in "modo parametrizzato" ma, in un certo senso, viene sottoposto a "upcast" all'oggetto **TwoTuple** non parametrizzato. Se volete tuttavia provare a intercettare il risultato di **f2()** in un oggetto **TwoTuple** parametrizzato, il compilatore visualizzerà un messaggio di tipo *warning*.

**Esercizio 15** (1) Verificate l'affermazione precedente.

**Esercizio 16** (2) Aggiunge un oggetto **SixTuple** a **Tuple.java** e testatelo in **TupleTest2.java**.

## *Un programma di utilità per Set*

Come ulteriore esempio dell'utilizzo dei metodi generici, considerate le relazioni matematiche che possono essere espresse servendosi dei **Set**. In modo

molto pratico, tali relazioni sono definibili sotto forma di metodi generici per essere utilizzate con tutti i diversi tipi:

```
//: net/mindview/util/Sets.java
package net.mindview.util;
import java.util.*;

public class Sets {
  public static <T> Set<T> union(Set<T> a, Set<T> b) {
    Set<T> result = new HashSet<T>(a);
    result.addAll(b);
    return result;
  }
  public static <T>
  Set<T> intersection(Set<T> a, Set<T> b) {
    Set<T> result = new HashSet<T>(a);
    result.retainAll(b);
    return result;
  }
  // Sottrae il sottoinsieme dal sovrainsieme:
  public static <T> Set<T>
  difference(Set<T> superset, Set<T> subset) {
    Set<T> result = new HashSet<T>(superset);
    result.removeAll(subset);
    return result;
  }
  // Riflessivo - tutto cio' che non e' incluso
  // nell'intersezione:
  public static <T> Set<T> complement(Set<T> a, Set<T> b) {
    return difference(union(a, b), intersection(a, b));
  }
} ///:~
```

I primi tre metodi duplicano il primo argomento copiandone i riferimenti in un nuovo oggetto **HashSet**, in modo che i **Set** passati come argomento non vengano direttamente modificati: il valore di ritorno è pertanto un nuovo oggetto **Set**.

I quattro metodi rappresentano le quattro operazioni in insiemistica: **union()** restituisce un **Set** contenente la combinazione dei due insiemi passati come argomento, **intersection()** restituisce un **Set** che contiene gli elementi comuni ai due argomenti, **difference()** esegue una sottrazione degli elementi del sottoinsieme (**subset**) da quelli del sovrainsieme (**superset**) e **complement()** restituisce un **Set** di tutti gli elementi che non figurano nell'intersezione. Per creare un semplice esempio che illustra gli effetti di questi metodi, considerate la seguente **enum**, che include i vari nomi inglesi dei colori pastello:

```
//: generics/watercolors/Watercolors.java
package generics.watercolors;

public enum Watercolors {
  ZINC, LEMON_YELLOW, MEDIUM_YELLOW, DEEP_YELLOW, ORANGE,
  BRILLIANT_RED, CRIMSON, MAGENTA, ROSE_MADDER, VIOLET,
  CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE,
  COBALT_BLUE_HUE, PERMANENT_GREEN, VIRIDIAN_HUE,
  SAP_GREEN, YELLOW_OCHRE, BURNT_SIENNA, RAW_UMBER,
  BURNT_UMBER, PAYNES_GRAY, IVORY_BLACK
} ///:~
```

Per ragioni pratiche, al fine di evitare che tutti i metodi debbano essere qualificati, l'**enum Watercolors** viene importata con **static**. Questo esempio si serve di **EnumSet**, un'utility di Java SE5 che consente di creare facilmente **Set** dalle **enum**: vedrete come utilizzare la classe **EnumSet** nel Capitolo 7. In questo caso, al metodo **static EnumSet.range()** vengono passati il primo e ultimo elemento dell'intervallo da creare nel **Set** risultante:

```
//: generics/WatercolorSets.java
import generics.watercolors.*;
import java.util.*;
import static net.mindview.util.Print.*;
import static net.mindview.util.Sets.*;
import static generics.watercolors.Watercolors.*;

public class WatercolorSets {
  public static void main(String[] args) {
    Set<Watercolors> set1 =
      EnumSet.range(BRILLIANT_RED, VIRIDIAN_HUE);
    Set<Watercolors> set2 =
```

```
      EnumSet.range(CERULEAN_BLUE_HUE, BURNT_UMBER);
    print("set1: " + set1);
    print("set2: " + set2);
    print("union(set1, set2: " + union(set1, set2));
    Set<Watercolors> subset = intersection(set1, set2);
    print("intersection(set1, set2): " + subset);
    print("difference(set1, subset): " +
      difference(set1, subset));
    print("difference(set2, subset): " +
      difference(set2, subset));
    print("complement(set1, set2): " +
      complement(set1, set2));
  }
} /* Output: (Esempio)
set1: [BRILLIANT_RED, CRIMSON, MAGENTA, ROSE_MADDER,
VIOLET, CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE,
COBALT_BLUE_HUE, PERMANENT_GREEN, VIRIDIAN_HUE]
set2: [CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE,
COBALT_BLUE_HUE, PERMANENT_GREEN, VIRIDIAN_HUE, SAP_GREEN,
YELLOW_OCHRE, BURNT_SIENNA, RAW_UMBER, BURNT_UMBER]
union(set1, set2): [SAP_GREEN, ROSE_MADDER, YELLOW_OCHRE,
PERMANENT_GREEN, BURNT_UMBER, COBALT_BLUE_HUE, VIOLET,
BRILLIANT_RED, RAW_UMBER, ULTRAMARINE, BURNT_SIENNA,
CRIMSON, CERULEAN_BLUE_HUE, PHTHALO_BLUE, MAGENTA,
VIRIDIAN_HUE]
intersection(set1, set2): [ULTRAMARINE, PERMANENT_GREEN,
COBALT_BLUE_HUE, PHTHALO_BLUE, CERULEAN_BLUE_HUE,
VIRIDIAN_HUE]
difference(set1, subset): [ROSE_MADDER, CRIMSON, VIOLET,
MAGENTA, BRILLIANT_RED]
difference(set2, subset): [RAW_UMBER, SAP_GREEN,
YELLOW_OCHRE, BURNT_SIENNA, BURNT_UMBER]
complement(set1, set2): [SAP_GREEN, ROSE_MADDER,
YELLOW_OCHRE, BURNT_UMBER, VIOLET, BRILLIANT_RED,
RAW_UMBER, BURNT_SIENNA, CRIMSON, MAGENTA]
*///:~
```

L'output mostra i risultati di ogni operazione.

Il seguente esempio utilizza **Sets.difference()** per evidenziare le differenze dei metodi tra le classi **Collection** e **Map** in **java.util**:

```
//: net/mindview/util/ContainerMethodDifferences.java
package net.mindview.util;
import java.lang.reflect.*;
import java.util.*;

public class ContainerMethodDifferences {
  static Set<String> methodSet(Class<?> type) {
    Set<String> result = new TreeSet<String>();
    for(Method m : type.getMethods())
      result.add(m.getName());
    return result;
  }
  static void interfaces(Class<?> type) {
    System.out.print("Interfaces in " +
      type.getSimpleName() + ": ");
    List<String> result = new ArrayList<String>();
    for(Class<?> c : type.getInterfaces())
      result.add(c.getSimpleName());
    System.out.println(result);
  }
  static Set<String> object = methodSet(Object.class);
  static { object.add("clone"); }
  static void
  difference(Class<?> superset, Class<?> subset) {
    System.out.print(superset.getSimpleName() +
      " extends " + subset.getSimpleName() + ", adds: ");
    Set<String> comp = Sets.difference(
      methodSet(superset), methodSet(subset));
    comp.removeAll(object); // Non mostra i metodi 'Object'
    System.out.println(comp);
    interfaces(superset);
  }
  public static void main(String[] args) {
```

```
    System.out.println("Collection: " +
      methodSet(Collection.class));
    interfaces(Collection.class);
    difference(Set.class, Collection.class);
    difference(HashSet.class, Set.class);
    difference(LinkedHashSet.class, HashSet.class);
    difference(TreeSet.class, Set.class);
    difference(List.class, Collection.class);
    difference(ArrayList.class, List.class);
    difference(LinkedList.class, List.class);
    difference(Queue.class, Collection.class);
    difference(PriorityQueue.class, Queue.class);
    System.out.println("Map: " + methodSet(Map.class));
    difference(HashMap.class, Map.class);
    difference(LinkedHashMap.class, HashMap.class);
    difference(SortedMap.class, Map.class);
    difference(TreeMap.class, Map.class);
  }
} ///:~
```

L'output di questo programma è già stato analizzato nel riepilogo del Volume 1, Capitolo 11.

**Esercizio 17** (4) Studiate la documentazione JDK per **EnumSet**, nella quale troverete la definizione del metodo **clone()**. Tuttavia, non è possibile eseguire **clone()** dal riferimento all'interfaccia **Set** passata in **Sets.java**. Siete in grado di modificare **Sets.java** per gestire il caso generale di un'interfaccia **Set** (come indicato negli esempi precedenti) e il caso speciale di un **EnumSet** utilizzando il metodo **clone()**, invece di creare un nuovo **HashSet**?

## Classi interne anonime

I generici sono utilizzabili anche con le classi interne e quelle interne anonime, come mostra l'esempio seguente che implementa l'interfaccia **Generator** servendosi di classi interne anonime:

```
//: generics/BankTeller.java
// Una semplice simulazione di cassa automatica.
```

```
import java.util.*;
import net.mindview.util.*;

class Customer {
  private static long counter = 1;
  private final long id = counter++;
  private Customer() {}
  public String toString() { return "Customer " + id; }
  // Un metodo per produrre oggetti Generator:
  public static Generator<Customer> generator() {
    return new Generator<Customer>() {
      public Customer next() { return new Customer(); }
    };
  }
}

class Teller {
  private static long counter = 1;
  private final long id = counter++;
  private Teller() {}
  public String toString() { return "Teller " + id; }
  // Un solo oggetto Generator:
  public static Generator<Teller> generator =
    new Generator<Teller>() {
      public Teller next() { return new Teller(); }
    };
}

public class BankTeller {
  public static void serve(Teller t, Customer c) {
    System.out.println(t + " serves " + c);
  }
  public static void main(String[] args) {
    Random rand = new Random(47);
    Queue<Customer> line = new LinkedList<Customer>();
    Generators.fill(line, Customer.generator(), 15);
    List<Teller> tellers = new ArrayList<Teller>();
```

```
    Generators.fill(tellers, Teller.generator, 4);
    for(Customer c : line)
      serve(tellers.get(rand.nextInt(tellers.size())), c);
  }
} /* Output:
Teller 3 serves Customer 1
Teller 2 serves Customer 2
Teller 3 serves Customer 3
Teller 1 serves Customer 4
Teller 1 serves Customer 5
Teller 3 serves Customer 6
Teller 1 serves Customer 7
Teller 2 serves Customer 8
Teller 3 serves Customer 9
Teller 3 serves Customer 10
Teller 2 serves Customer 11
Teller 4 serves Customer 12
Teller 2 serves Customer 13
Teller 1 serves Customer 14
Teller 1 serves Customer 15
*///:~
```

**Customer** e **Teller** hanno costruttori di tipo **private**, che vi costringono a servirvi di oggetti **Generator**. La classe **Customer** dispone di un metodo **generator()** che a ogni chiamata produce un nuovo oggetto **Generator<Customer>**. Potreste non avere bisogno di oggetti **Generator** multipli: in questo caso, in effetti, **Teller** crea un solo oggetto **generator** pubblico. Potete vedere entrambi gli approcci utilizzati nei metodi **fill()** di **main()**.

Dal momento che sia il metodo **generator()** in **Customer** sia l'oggetto **Generator** in **Teller** sono **static**, non possono fare parte di un'interfaccia; di conseguenza è impossibile "generalizzare" questa sintassi specifica, che tuttavia funziona piuttosto bene con il metodo **fill()**.

Nel Volume 3, Capitolo 1 avrete modo di esaminare altre varianti di questo problema di accodamento.

**Esercizio 18** (3) Sul modello di **BankTeller.java**, create un esempio in cui un grande pesce (**BigFish**) mangia un pesciolino (**LittleFish**) nell'oceano (**Ocean**).

## Costruzione di modelli complessi

Un beneficio importante offerto dai generici è la possibilità di creare modelli complessi in modo semplice e sicuro. Per esempio, potete creare facilmente una **List** di tuple:

```
//: generics/TupleList.java
// Combina i tipi generici per costruire tipi generici
// complessi.
import java.util.*;
import net.mindview.util.*;

public class TupleList<A,B,C,D>
extends ArrayList<FourTuple<A,B,C,D>> {
  public static void main(String[] args) {
    TupleList<Vehicle, Amphibian, String, Integer> tl =
      new TupleList<Vehicle, Amphibian, String, Integer>();
    tl.add(TupleTest.h());
    tl.add(TupleTest.h());
    for(FourTuple<Vehicle,Amphibian,String,Integer> i: tl)
      System.out.println(i);
  }
} /* Output: (75% match)
(Vehicle@be2358, Amphibian@27b4d, hi, 47)
(Vehicle@ed2ae8, Amphibian@9c26f5, hi, 47)
*///:~
```

Ecco un altro esempio che dimostra l'immediatezza della creazione di modelli complessi per mezzo dei tipi generici. Anche se ogni classe viene generata come elemento costitutivo, il totale ha molte parti. In questo caso il modello è una rivendita al dettaglio con corsie, scaffali e prodotti:

```
//: generics/Store.java
// Costruzione di un modello complesso mediante contenitori
// generici.
import java.util.*;
import net.mindview.util.*;

class Product {
```

```
  private final int id;
  private String description;
  private double price;
  public Product(int IDnumber, String descr, double price){
    id = IDnumber;
    description = descr;
    this.price = price;
    System.out.println(toString());
  }
  public String toString() {
    return id + ": " + description + ", price: $" + price;
  }
  public void priceChange(double change) {
    price += change;
  }
  public static Generator<Product> generator =
    new Generator<Product>() {
      private Random rand = new Random(47);
      public Product next() {
        return new Product(rand.nextInt(1000), "Test",
          Math.round(rand.nextDouble() * 1000.0) + 0.99);
      }
    };
}

class Shelf extends ArrayList<Product> {
  public Shelf(int nProducts) {
    Generators.fill(this, Product.generator, nProducts);
  }
}

class Aisle extends ArrayList<Shelf> {
  public Aisle(int nShelves, int nProducts) {
    for(int i = 0; i < nShelves; i++)
      add(new Shelf(nProducts));
  }
}
```

```
class CheckoutStand {}
class Office {}

public class Store extends ArrayList<Aisle> {
  private ArrayList<CheckoutStand> checkouts =
    new ArrayList<CheckoutStand>();
  private Office office = new Office();
  public Store(int nAisles, int nShelves, int nProducts) {
    for(int i = 0; i < nAisles; i++)
      add(new Aisle(nShelves, nProducts));
  }
  public String toString() {
    StringBuilder result = new StringBuilder();
    for(Aisle a : this)
      for(Shelf s : a)
        for(Product p : s) {
          result.append(p);
          result.append("\n");
        }
    return result.toString();
  }
  public static void main(String[] args) {
    System.out.println(new Store(14, 5, 10));
  }
} /* Output:
258: Test, price: $400.99
861: Test, price: $160.99
868: Test, price: $417.99
207: Test, price: $268.99
551: Test, price: $114.99
278: Test, price: $804.99
520: Test, price: $554.99
140: Test, price: $530.99
...
*///:~
```

Come potete vedere, in **Store.toString()** il risultato è costituito da molti livelli di contenitori, che sono tuttavia sicuri per quanto riguarda i tipi e facilmente gestibili. Quello che sorprende è che non è intellettualmente proibitivo assemblare un simile modello.

**Esercizio 19** (2) Sull'esempio di **Store.java**, sviluppate un modello di una nave portacontainer.

## Mistero della cancellazione

Familiarizzando via via con l'utilizzo dei generici, scoprirete un certo numero di dettagli che all'inizio vi sembreranno privi di senso: per esempio, sebbene possiate scrivere **ArrayList.class**, la sintassi **ArrayList<Integer>.class** non è ammessa. Considerate quanto segue:

```
//: generics/ErasedTypeEquivalence.java
import java.util.*;

public class ErasedTypeEquivalence {
  public static void main(String[] args) {
    Class c1 = new ArrayList<String>().getClass();
    Class c2 = new ArrayList<Integer>().getClass();
    System.out.println(c1 == c2);
  }
} /* Output:
true
*///:~
```

Ci sarebbe molto da ridire sul fatto che **ArrayList<String>** e **ArrayList<Integer>** sono tipi differenti. Tuttavia, tipi diversi si comportano in modo diverso e se provate, per esempio, a inserire un **Integer** in un **ArrayList<String>**, otterrete un comportamento differente (*errore*) da quello provocato dall'inserimento di un **Integer** in un **ArrayList<Integer>** (*riuscita*). Malgrado questo, il programma precedente suggerisce che i due oggetti siano dello stesso tipo.

Ecco un interessante tassello da aggiungere a questo puzzle:

```
//: generics/LostInformation.java
import java.util.*;
```

```
class Frob {}
class Fnorkle {}
class Quark<Q> {}
class Particle<POSITION,MOMENTUM> {}

public class LostInformation {
  public static void main(String[] args) {
    List<Frob> list = new ArrayList<Frob>();
    Map<Frob,Fnorkle> map = new HashMap<Frob,Fnorkle>();
    Quark<Fnorkle> quark = new Quark<Fnorkle>();
    Particle<Long,Double> p = new Particle<Long,Double>();
    System.out.println(Arrays.toString(
      list.getClass().getTypeParameters()));
    System.out.println(Arrays.toString(
      map.getClass().getTypeParameters()));
    System.out.println(Arrays.toString(
      quark.getClass().getTypeParameters()));
    System.out.println(Arrays.toString(
      p.getClass().getTypeParameters()));
  }
} /* Output:
[E]
[K, V]
[Q]
[POSITION, MOMENTUM]
*///:~
```

Secondo la documentazione JDK, **Class.getTypeParameters()** "restituisce un array di oggetti **TypeVariable** che rappresentano le variabili di tipo dichiarate nella dichiarazione generica...": questa affermazione sembra indicare che sia possibile determinare i tipi di parametro. Tuttavia, come potete vedere dall'output, tutto ciò che otterrete sono gli identificativi utilizzati come segnaposto per i parametri, un'informazione di per sé poco interessante.

La nuda verità è che:

> *Nel codice generico non è disponibile alcuna informazione sui tipi di parametro generici.*

Quindi, mentre potete conoscere dettagli quali l'identificativo del parametro di tipo e i limiti del tipo generico, non potete invece sapere quali sono i parametri di tipo effettivamente utilizzati nella creazione di una determinata istanza. Questo aspetto, particolarmente frustrante per chi proviene dal linguaggio C++, è il problema fondamentale con cui avete a che fare quando lavorate con i generici di Java.

I generici di Java vengono implementati mediante la cosiddetta *cancellazione* (*erasure*), che implica l'eliminazione di qualsiasi informazione specifica sul tipo nel momento in cui utilizzate un generico. All'interno del generico, l'unico dato di cui disponete è il fatto che state utilizzando un oggetto: di conseguenza, al momento dell'esecuzione **List<String>** e **List<Integer>** *sono effettivamente* dello stesso tipo, poiché entrambe le forme sono state "cancellate" e ripristinate al loro *tipo grezzo* originario, **List**. La comprensione del meccanismo di cancellazione e del modo per gestirlo sono due degli ostacoli principali che dovete affrontare nello studio dei generici di Java, e saranno l'argomento primario dei prossimi paragrafi.

## Approccio di C++

Questo è un esempio di codice C++ che utilizza i *modelli*, o *template*. Come potete vedere, la sintassi per i tipi parametrizzati è abbastanza simile, a ulteriore conferma che Java ha preso spunto dal linguaggio C++:

```
//: generics/Templates.cpp
#include <iostream>
using namespace std;

template<class T> class Manipulator {
  T obj;
public:
  Manipulator(T x) { obj = x; }
  void manipulate() { obj.f(); }
};

class HasF {
public:
  void f() { cout << "HasF::f()" << endl; }
};
```

```
int main() {
  HasF hf;
  Manipulator<HasF> manipulator(hf);
  manipulator.manipulate();
} /* Output:
HasF::f()
///:~
```

La classe **Manipulator** memorizza un oggetto di tipo **T**. È particolarmente interessante il metodo **manipulate()**, che chiama un metodo **f()** su **obj**: come può **manipulate()** sapere che per il parametro di tipo **T** è disponibile il metodo **f()**? Quando istanziate il modello il compilatore C++ esegue alcuni controlli, pertanto al momento di istanziare **Manipulator<HasF>** il compilatore sa già che **HasF** possiede un metodo **f()**. Se così non fosse otterreste infatti un errore di compilazione, a tutela della sicurezza dei tipi.

La scrittura di questo genere di codice in C++ è immediata perché, al momento di essere istanziato, il codice del modello conosce i tipi dei suoi parametri di template. I generici di Java si comportano in modo diverso, come dimostra la seguente "traduzione" in Java di **HasF**:

```
//: generics/HasF.java

public class HasF {
  public void f() { System.out.println("HasF.f()"); }
} ///:~
```

Se prendete il resto del codice e lo convertite in Java, esso non compilerà:

```
//: generics/Manipulation.java
// {CompileTimeError} (Non compilera')

class Manipulator<T> {
  private T obj;
  public Manipulator(T x) { obj = x; }
  // Errore: cannot find symbol: method f():
  public void manipulate() { obj.f(); }
}

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

```
    HasF hf = new HasF();
    Manipulator<HasF> manipulator =
      new Manipulator<HasF>(hf);
    manipulator.manipulate();
  }
} ///:~
```

A causa della cancellazione, il compilatore Java non è in grado di far corrispondere la necessità di **manipulate()** di chiamare **f()** su **obj** con il fatto che **HasF** dispone di un **f()**. Per chiamare **f()** dovete "aiutare" la classe generica, assegnandole un *limite* che indichi al compilatore di accettare soltanto i tipi conformi a tale limite. Per l'impostazione di questo limite si riutilizza la parola chiave **extends**, grazie alla quale il seguente codice compilerà senza problemi:

```
//: generics/Manipulator2.java

class Manipulator2<T extends HasF> {
  private T obj;
  public Manipulator2(T x) { obj = x; }
  public void manipulate() { obj.f(); }
} ///:~
```

Il limite **<T extends HasF>** indica che **T** deve essere di tipo **HasF** o di una classe derivata da **HasF**: se questo è vero, sarà possibile chiamare **f()** su **obj**.

Si è detto che un parametro di tipo generico *cancella fino al suo primo limite*: ricordate che possono esistere più limiti, come vedrete in seguito. Tuttavia bisognerebbe anche parlare della *cancellazione del parametro di tipo*, perché il compilatore, in realtà, sostituisce il parametro di tipo con la relativa cancellazione: pertanto, nel caso in esame, **T** cancella fino ad **HasF**, il che equivale a sostituire **T** con **HasF** nel corpo della classe.

Potreste osservare, a ragione, che in **Manipulation2.java** i generici non danno alcun contributo; in effetti, potreste facilmente eseguire voi stessi la cancellazione e creare una classe priva di generici:

```
//: generics/Manipulator3.java

class Manipulator3 {
  private HasF obj;
```

```
  public Manipulator3(HasF x) { obj = x; }
  public void manipulate() { obj.f(); }
} ///:~
```

Questo mette in risalto una considerazione essenziale: i generici sono utili soltanto quando occorra servirsi di parametri di tipo che siano più "generici" di un tipo specifico e di tutti i suoi sottotipi, in pratica quando desiderate che il codice funzioni su classi diverse. Di conseguenza, i parametri di tipo e la loro applicazione nel codice generico a scopi pratici saranno solitamente più complessi della semplice sostituzione di classe. Tuttavia, non si può semplicemente affermare che tutto ciò che è nella forma **<T extends HasF>** debba essere invalidato. Per esempio, se una classe ha un metodo che restituisce **T**, allora i generici torneranno utili perché restituiranno il tipo esatto:

```
//: generics/ReturnGenericType.java

class ReturnGenericType<T extends HasF> {
  private T obj;
  public ReturnGenericType(T x) { obj = x; }
  public T get() { return obj; }
} ///:~
```

Dovete esaminare tutto il codice e capire se è "abbastanza complesso" da giustificare l'utilizzo dei generici. Nel prosieguo del capitolo vedrete il meccanismo dei limiti in maggiore dettaglio.

> **Esercizio 20** (1) Create un'interfaccia con due metodi e una classe che implementi tale interfaccia aggiungendovi un altro metodo. In un'altra classe, create un metodo generico con un tipo di argomento limitato dall'interfaccia e dimostrate che i metodi nell'interfaccia sono richiamabili dall'interno di questo metodo generico. In **main()**, passate un'istanza della classe implementante l'interfaccia al metodo generico.

## *Compatibilità in vista della migrazione*

Per cercare di dissipare tutta la confusione che potrebbe avervi creato il concetto di cancellazione dovete capire chiaramente che non si tratta di una caratteristica del linguaggio, bensì di un compromesso nell'implementazione dei generici di Java, necessaria perché i generici non sono stati resi parte del linguaggio fin dall'inizio. Questo compromesso vi causerà parecchi inconve-

nienti, quindi è opportuno che vi ci abituiate rapidamente e ne comprendiate le ragioni.

Se i generici fossero stati inclusi in Java 1.0 tale funzionalità non sarebbe stata implementata mediante la cancellazione, ma si sarebbe fatto utilizzo della cosiddetta *reificazione* per mantenere i parametri di tipo come entità "di prima classe": in tal modo, avreste beneficiato di un linguaggio basato sui tipi e di operazioni riflessive sui parametri di tipo. Come vedrete nel prosieguo del capitolo, la cancellazione riduce la "genericità" dei generici, che continuano a essere utili in Java anche se non così utili come potrebbero essere; il motivo è appunto la cancellazione.

In un'implementazione basata sulla cancellazione, i tipi generici sono considerati "di seconda classe" e non possono essere utilizzati in alcuni importanti contesti. I tipi generici sono presenti soltanto durante il controllo di tipo statico, dopodiché ogni tipo generico presente nel programma viene cancellato e sostituito con un limite superiore non generico. Per esempio, annotazioni di tipo come **List<T>** vengono eliminate da **List** e le normali variabili di tipo cancellate da **Object**, a meno che non sia stato specificato un limite.

La motivazione primaria della cancellazione è che permette a client "generalizzati" di essere utilizzati con librerie non "generalizzate" e viceversa, secondo un comportamento spesso definito con l'espressione *migration compatibility*. In un mondo ideale, un giorno tutto sarebbe stato generalizzato nello stesso momento; nella realtà, invece, anche se i programmatori scrivessero soltanto codice generico, dovrebbero comunque tenere conto di librerie non generiche scritte prima di Java SE5. Gli autori di queste librerie potrebbero non essere stimolati in alcun modo a generalizzare il loro codice, oppure impiegarci molto tempo.

Per questi motivi, i generici di Java devono non soltanto garantire la *retrocompatibilità*, in accordo alla quale il codice e i file di classe esistenti sono ancora legali e dovranno mantenere il loro significato originale, ma anche supportare la compatibilità in vista della migrazione (*migration compatibility*), in modo che le librerie possano essere rese generiche secondo le esigenze effettive e che, una volta diventate generiche, non provochino errori nel codice e nelle applicazioni. Dopo avere deciso che questo era l'obiettivo, i progettisti di Java e i vari gruppi che lavoravano al problema son giunti alla conclusione che la cancellazione era l'unica soluzione fattibile: la cancellazione consente questa "migrazione" verso i generici permettendo la coesistenza di codice generico e di codice non generico.

Supponete, per esempio, di avere un'applicazione che utilizzi due librerie, **X** e **Y**, e che **Y** utilizzi la libreria **Z**. Con l'avvento di Java SE5, gli sviluppatori di questa applicazione e delle librerie vorranno probabilmente adattarle

ai generici; ognuno di loro, tuttavia, avrà motivazioni e vincoli diversi per quanto riguarda i tempi di migrazione. Per ragioni di compatibilità, le due librerie e l'applicazione devono essere indipendenti le une dall'altra per quanto riguarda l'utilizzo dei generici, quindi non devono poter rilevare se altre librerie stanno utilizzando i generici o meno: la prova che una determinata libreria utilizza i generici, quindi, deve essere "cancellata".

Senza l'implementazione di una forma di "percorso di migrazione", tutte le librerie che sono state sviluppate nel corso degli anni rischierebbero di essere ignorate dagli sviluppatori che scelgono di implementare i generici di Java. Le librerie sono indiscutibilmente il componente dei linguaggi di programmazione che ha il maggiore impatto sulla produttività, e la scelta di ignorarle rappresenta quindi un costo impossibile da sostenere. Solo il tempo potrà confermare se la cancellazione si rivelerà il percorso di migrazione migliore, o semplicemente l'unico possibile.

## *Problema della cancellazione*

Avete visto che le giustificazioni primarie per la cancellazione sono il processo di transizione da codice non generalizzato a quello generalizzato e l'incorporazione dei generici nel linguaggio senza effetti negativi sulle librerie esistenti. La cancellazione consente di continuare a utilizzare il codice non generico attuale del client senza operare cambiamenti, fino al momento in cui i programmatori client non siano pronti a riscrivere il codice per i generici. Questa è una motivazione notevole: grazie alla cancellazione, si evita che all'improvviso il codice smetta di funzionare.

L'onere legato alla cancellazione è notevole. I tipi generici non possono essere utilizzati per funzionalità che fanno esplicito riferimento ai tipi utilizzabili a runtime, quali i cast, le operazioni di **instanceof** e le espressioni **new**. Poiché per i parametri tutte le informazioni sui tipi vengono perse, ogni volta che scriverete codice generico dovrete ricordarvi che le informazioni di cui disporrete su un parametro sono soltanto "apparenti". Quindi, se scrivete codice come

```
class Foo<T> {
  T var;
}
```

"sembra" che quando generate un'istanza di **Foo**:

```
Foo<Cat> f = new Foo<Cat>();
```

il codice nella classe **Foo** debba sapere che in quel momento sta funzionando con un **Cat**. La sintassi suggerisce fortemente che il tipo **T** sia in fase di sostituzione in tutto il codice della classe, ma non è così. Dovrete pertanto ricordarvi, ogni volta che scriverete codice per la classe, che si tratta di un semplice **Object**.

Inoltre, la cancellazione e la compatibilità in vista della migrazione implicano che l'utilizzo dei generici non venga fatto rispettare quando vi aspettereste che lo fosse:

```
//: generics/ErasureAndInheritance.java

class GenericBase<T> {
  private T element;
  public void set(T arg) { arg = element; }
  public T get() { return element; }
}

class Derived1<T> extends GenericBase<T> {}

class Derived2 extends GenericBase {} // Nessun avviso del
                                      // compilatore

// class Derived3 extends GenericBase<?> {}
// Uno strano errore:
//   unexpected type found : ?
//   required: class or interface without bounds

public class ErasureAndInheritance {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    Derived2 d2 = new Derived2();
    Object obj = d2.get();
    d2.set(obj); // Avviso del compilatore!
  }
} ///:~
```

**Derived2** eredita da **GenericBase** senza i parametri generici, ciononostante il compilatore non visualizza alcun avvertimento: l'avvertimento verrà sollevato soltanto al momento della chiamata a **set()**.

Per disabilitare l'avvertimento del compilatore, Java mette a disposizione l'annotazione che vedete nel codice, presente tuttavia soltanto a partire da Java SE5:

```
@SuppressWarnings("unchecked")
```

Notate che questa istruzione è posta nell'ambito di visibilità del metodo che genera l'avvertimento, anziché dell'intera classe. Quando disabilitate un avvertimento è bene che lo facciate nel modo più "accurato" possibile, in modo da evitare i problemi che deriverebbero da un'eccessiva "disinvoltura" nella disattivazione.

L'errore prodotto da **Derived3** significa probabilmente che il compilatore si aspetta una classe di base "grezza".

Se a tutto questo aggiungete l'impegno supplementare richiesto dal trattamento dei limiti quando occorre gestire il parametro tipo come qualcosa di più di un semplice **Object**, vi renderete conto che la gestione dei tipi parametrizzati in Java presenta decisamente più inconvenienti che vantaggi, se paragonata con la stessa funzionalità di linguaggi quali C++, Ada o Eiffel. Questo non significa che tali linguaggi vi garantiscano più di quanto Java offre per risolvere la maggior parte dei problemi di programmazione, quanto piuttosto che i meccanismi per i tipi parametrizzati di questi linguaggi sono più flessibili e potenti.

## *Azione sui limiti*

A causa della cancellazione, molti ritengono che l'aspetto più caotico dei generici risieda nel fatto che consentono di rappresentare entità prive di significato. Per esempio:

```
//: generics/ArrayMaker.java
import java.lang.reflect.*;
import java.util.*;

public class ArrayMaker<T> {
  private Class<T> kind;
  public ArrayMaker(Class<T> kind) { this.kind = kind; }
  @SuppressWarnings("unchecked")
  T[] create(int size) {
    return (T[])Array.newInstance(kind, size);
  }
```

```
    public static void main(String[] args) {
      ArrayMaker<String> stringMaker =
        new ArrayMaker<String>(String.class);
      String[] stringArray = stringMaker.create(9);
      System.out.println(Arrays.toString(stringArray));
    }
  } /* Output:
  [null, null, null, null, null, null, null, null, null]
  *///:~
```

Anche se **kind** è registrato come **Class<T>**, la cancellazione fa sì che in realtà sia memorizzato soltanto come una **Class**, senza parametri; pertanto quando ve ne servirete, per esempio nella generazione di un array, **Array.newInstance()** non avrà realmente le informazioni di tipo implicite in **kind** e di conseguenza non potrà fornire il risultato specificato. Per questo motivo tale risultato dovrà essere sottoposto a cast, il che produrrà un avvertimento che non sarete in grado di soddisfare.

Ricordate che l'utilizzo del metodo **Array.newInstance()** è la tecnica consigliata per creare gli array con i generici.

Se generate un contenitore, invece, tutto cambia:

```
//: generics/ListMaker.java
import java.util.*;

public class ListMaker<T> {
  List<T> create() { return new ArrayList<T>(); }
  public static void main(String[] args) {
    ListMaker<String> stringMaker= new ListMaker<String>();
    List<String> stringList = stringMaker.create();
  }
} ///:~
```

Il compilatore non produce alcun avvertimento, anche se sapete (in base alla cancellazione) che la **<T>** in **new ArrayList<T>()** all'interno di **create()** viene rimossa: al momento dell'esecuzione la **<T>** non esiste all'interno della classe, pertanto non avrebbe senso che vi rimanesse. Se invece seguite questa idea e modificate l'espressione in **new ArrayList()**, il compilatore vi avvertirà.

Il comportamento del compilatore è davvero insignificante in questo caso? Immaginate di includere alcuni oggetti in **list** prima della restituzione, come in questo esempio:

```
//: generics/FilledListMaker.java
import java.util.*;

public class FilledListMaker<T> {
  List<T> create(T t, int n) {
    List<T> result = new ArrayList<T>();
    for(int i = 0; i < n; i++)
      result.add(t);
    return result;
  }
  public static void main(String[] args) {
    FilledListMaker<String> stringMaker =
      new FilledListMaker<String>();
    List<String> list = stringMaker.create("Hello", 4);
    System.out.println(list);
  }
} /* Output:
[Hello, Hello, Hello, Hello]
*///:~
```

Malgrado il compilatore non possa conoscere nulla a proposito della **T** all'interno di **create()**, in fase di compilazione può comunque garantire che quanto inserirete in **result** sia di tipo **T**, e quindi conforme ad **ArrayList<T>**. Pertanto, anche se la cancellazione rimuove le informazioni sul tipo reale all'interno di un metodo o di una classe, il compilatore può sempre garantire la coerenza interna nel modo in cui il tipo viene utilizzato nell'ambito del metodo o della classe.

Poiché la cancellazione rimuove le informazioni di tipo nel corpo di un metodo, ciò che conta al momento dell'esecuzione sono i limiti (*boundary*), vale a dire i punti in cui gli oggetti entrano ed escono dal metodo. Questi sono i punti in cui il compilatore esegue i controlli sul tipo e inserisce il codice di casting. Considerate l'esempio seguente non generico:

```
//: generics/SimpleHolder.java

public class SimpleHolder {
```

```
    private Object obj;
    public void set(Object obj) { this.obj = obj; }
    public Object get() { return obj; }
    public static void main(String[] args) {
      SimpleHolder holder = new SimpleHolder();
      holder.set("Item");
      String s = (String)holder.get();
    }
  } ///:~
```

Se decompilate il risultato con **javap - c SimpleHolder**, dopo qualche operazione di editing otterrete:

```
public void set(java.lang.Object);
   0:    aload_0
   1:    aload_1
   2:    putfield #2; //Field obj:Object;
   5:    return

public java.lang.Object get();
   0:    aload_0
   1:    getfield #2; //Field obj:Object;
   4:    areturn

public static void main(java.lang.String[]);
   0:    new #3; //class SimpleHolder
   3:    dup
   4:    invokespecial #4; //Method "<init>":()V
   7:    astore_1
   8:    aload_1
   9:    ldc #5; //String Item
   11:   invokevirtual #6; //Method set:(Object;)V
   14:   aload_1
   15:   invokevirtual #7; //Method get:()Object;
   18:   checkcast #8; //class java/lang/String
   21:   astore_2
   22:   return
```

I metodi **set()** e **get()** si limitano a memorizzare e produrre il valore, e il cast viene controllato al punto di chiamata a **get()**.

Ora incorporate i generici nel codice precedente:

```
//: generics/GenericHolder.java

public class GenericHolder<T> {
  private T obj;
  public void set(T obj) { this.obj = obj; }
  public T get() { return obj; }
  public static void main(String[] args) {
    GenericHolder<String> holder =
      new GenericHolder<String>();
    holder.set("Item");
    String s = holder.get();
  }
} ///:~
```

Non solo è scomparsa l'esigenza del cast da **get()**, ma ora sapete anche che il valore passato a **set()** viene controllato in fase di compilazione. Analizzate i bytecode relativi:

```
public void set(java.lang.Object);
   0:     aload_0
   1:     aload_1
   2:     putfield #2; //Field obj:Object;
   5:     return

public java.lang.Object get();
   0:     aload_0
   1:     getfield #2; //Field obj:Object;
   4:     areturn
public static void main(java.lang.String[]);
   0:     new #3; //class GenericHolder
   3:     dup
   4:     invokespecial #4; //Method "<init>":()V
   7:     astore_1
   8:     aload_1
```

```
9:    ldc #5; //String Item
11:   invokevirtual #6; //Method set:(Object;)V
14:   aload_1
15:   invokevirtual #7; //Method get:()Object;
18:   checkcast #8; //class java/lang/String
21:   astore_2
22:   return
```

Il codice risultante è identico. L'attività supplementare di controllo del tipo in entrata in **set()** non è a carico del programma, essendo eseguita dal compilatore. È ancora necessario procedere al cast del valore in uscita da **get()**, un'attività tuttavia non più onerosa di quella che eseguireste voi stessi, che viene condotta automaticamente dal compilatore e produce codice molto più semplice da scrivere e leggere.

Dal momento che i metodi **get()** e **set()** producono bytecode simili, tutte le azioni che coinvolgono i generici avvengono sui limiti, sia il controllo aggiuntivo in sede di compilazione sia il cast applicato ai valori in uscita. Per fare chiarezza sull'argomento della cancellazione, vi sarà di aiuto ricordare che tutte le azioni avvengono sui limiti.

## Compensazione della cancellazione

Come avete visto, la cancellazione causa la perdita della capacità di eseguire determinate operazioni con il codice generico. Qualunque cosa richieda la conoscenza del tipo esatto al momento dell'esecuzione, non funzionerà:

```
//: generics/Erased.java
// {CompileTimeError} (Non compilera')

public class Erased<T> {
  private final int SIZE = 100;
  public static void f(Object arg) {
    if(arg instanceof T) {}          // Errore
    T var = new T();                 // Errore
    T[] array = new T[SIZE];         // Errore
    T[] array = (T)new Object[SIZE]; /* Avvertimento non
                                        controllato */
  }
} ///:~
```

Talvolta riuscirete a ovviare a questi problemi per mezzo della programmazione, ma a volte dovrete compensare la cancellazione introducendo un cosiddetto "tag di tipo" (*type tag*): in pratica, dovrete passare esplicitamente nell'oggetto **Class** l'indicazione del tipo, per poterlo utilizzare nelle espressioni di tipo.

Per esempio, il tentativo di utilizzare **instanceof** nel programma precedente fallisce perché l'informazione sul tipo è stata cancellata. Se introducete un tag di tipo, potrete servirvi di un metodo **isInstance()** dinamico:

```
//: generics/ClassTypeCapture.java

class Building {}
class House extends Building {}

public class ClassTypeCapture<T> {
  Class<T> kind;
  public ClassTypeCapture(Class<T> kind) {
    this.kind = kind;
  }
  public boolean f(Object arg) {
    return kind.isInstance(arg);
  }
  public static void main(String[] args) {
    ClassTypeCapture<Building> ctt1 =
      new ClassTypeCapture<Building>(Building.class);
    System.out.println(ctt1.f(new Building()));
    System.out.println(ctt1.f(new House()));
    ClassTypeCapture<House> ctt2 =
      new ClassTypeCapture<House>(House.class);
    System.out.println(ctt2.f(new Building()));
    System.out.println(ctt2.f(new House()));
  }
} /* Output:
true
true
false
true
*///:~
```

Il compilatore si incarica di verificare che il tag di tipo corrisponda all'argomento generico.

> **Esercizio 21** (4) Modificate **ClassTypeCapture.java** aggiungendo una mappa **Map<String,Class<?>>** e i metodi **addType(String typename, Class<?> kind)** e **createNew(String typename)**. Il metodo **createNew()** produrrà una nuova istanza della classe associata con la relativa stringa di argomento, oppure visualizzerà un messaggio d'errore.

## *Creazione di istanze di tipi*

Il tentativo di generare un nuovo **T()** in **Erased.java** non funzionerà, in parte a causa della cancellazione e in parte perché il compilatore non può verificare che la **T** abbia un costruttore predefinito, ovvero privo di argomenti. In C++, però, questa operazione è naturale, diretta e sicura perché controllata in fase di compilazione:

```
//: generics/InstantiateGenericType.cpp
// E' C++, non Java!

template<class T> class Foo {
  T x; // Create un campo di tipo T
  T* y; // Puntatore a T
public:
  // Inizializza il puntatore:
  Foo() { y = new T(); }
};

class Bar {};

int main() {
  Foo<Bar> fb;
  Foo<int> fi; // ...e funziona con i primitivi
} ///:~
```

La soluzione Java richiede di passare un oggetto factory e di servirsene per costruire la nuova istanza.

Un oggetto factory molto pratico è appunto **Class**, quindi se utilizzate un tag di tipo potrete servirvi di **newInstance()** per creare un nuovo oggetto di quel tipo:

```
//: generics/InstantiateGenericType.java
import static net.mindview.util.Print.*;

class ClassAsFactory<T> {
  T x;
  public ClassAsFactory(Class<T> kind) {
    try {
      x = kind.newInstance();
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
}

class Employee {}

public class InstantiateGenericType {
  public static void main(String[] args) {
    ClassAsFactory<Employee> fe =
      new ClassAsFactory<Employee>(Employee.class);
    print("ClassAsFactory<Employee> succeeded");
    try {
      ClassAsFactory<Integer> fi =
        new ClassAsFactory<Integer>(Integer.class);
    } catch(Exception e) {
      print("ClassAsFactory<Integer> failed");
    }
  }
} /* Output:
ClassAsFactory<Employee> succeeded
ClassAsFactory<Integer> failed
*///:~
```

Il codice compila, ma l'output "**ClassAsFactory<Integer> failed**" mostra che il codice non funziona come vi aspettereste, poiché **Integer** non possiede un costruttore predefinito. Poiché l'errore non viene intercettato in sede di compilazione, questo approccio è da considerarsi disapprovato: i tecnici Sun consigliano invece di utilizzare una factory esplicita e di limitare il tipo, in modo che accetti soltanto una classe che implementa questa factory:

```
//: generics/FactoryConstraint.java

interface FactoryI<T> {
  T create();
}

class Foo2<T> {
  private T x;
  public <F extends FactoryI<T>> Foo2(F factory) {
    x = factory.create();
  }
  // ...
}

class IntegerFactory implements FactoryI<Integer> {
  public Integer create() {
    return new Integer(0);
  }
}

class Widget {
  public static class Factory implements FactoryI<Widget> {
    public Widget create() {
      return new Widget();
    }
  }
}

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

```
    new Foo2<Integer>(new IntegerFactory());
    new Foo2<Widget>(new Widget.Factory());
  }
} ///:~
```

Notate che questa non è altro che una variazione del passaggio di **Class<T>**. Entrambe le tecniche passano oggetti factory; **Class<T>** è l'oggetto factory nativo, mentre il metodo appena illustrato genera un oggetto factory esplicito offrendovi tuttavia il controllo in fase di compilazione.

Un altro approccio è il design pattern *Template Method*. Nell'esempio seguente **get()** è il metodo Template, mentre **create()** è definito nella sottoclasse per produrre un oggetto di quel tipo:

```
//: generics/CreatorGeneric.java

abstract class GenericWithCreate<T> {
  final T element;
  GenericWithCreate() { element = create(); }
  abstract T create();
}

class X {}

class Creator extends GenericWithCreate<X> {
  X create() { return new X(); }
  void f() {
    System.out.println(element.getClass().getSimpleName());
  }
}
public class CreatorGeneric {
  public static void main(String[] args) {
    Creator c = new Creator();
    c.f();
  }
} /* Output:
X
*///:~
```

**Esercizio 22** (6) Utilizzate un tag di tipo con la riflessione per creare un metodo che impiega la versione con argomenti di **newInstance()** per generare un oggetto di una classe con un costruttore non predefinito.

**Esercizio 23** (1) Modificate **FactoryConstraint.java** in modo che **create()** accetti un argomento.

**Esercizio 24** (3) Modificate l'Esercizio 21 in modo che gli oggetti factory siano memorizzati in **Map**, anziché in **Class**.

## *Array di generici*

Come si è visto in **Erased.java**, non è possibile creare array di generici. La soluzione generale consiste nell'utilizzo di un **ArrayList** in ogni situazione in cui vorreste creare un array:

```
//: generics/ListOfGenerics.java
import java.util.*;

public class ListOfGenerics<T> {
  private List<T> array = new ArrayList<T>();
  public void add(T item) { array.add(item); }
  public T get(int index) { return array.get(index); }
} ///:~
```

In questo caso, otterrete il comportamento di un array, ma con il livello di sicurezza in sede di compilazione che viene garantito dai generici.

Talvolta potreste comunque volere creare un array di tipi generici: tenete presente, a questo proposito, che **ArrayList** internamente si serve proprio degli array. Curiosamente, Java permette di definire un *riferimento* in maniera tale che non darà problemi con il compilatore. Per esempio:

```
//: generics/ArrayOfGenericReference.java

class Generic<T> {}

public class ArrayOfGenericReference {
  static Generic<Integer>[] gia;
} ///:~
```

Il compilatore accetta questo codice senza segnalare nessun avvertimento; in ogni caso, non avrete mai la possibilità di creare un array di quel tipo esatto (inclusi i parametri di tipo), pertanto questo modo di procedere genera una certa confusione. Poiché tutti gli array hanno la stessa struttura (dimensioni delle singole posizioni e struttura dell'array) a prescindere dal tipo che contengono, dovrebbe essere possibile creare un array di **Object** da sottoporre poi a cast al tipo di array desiderato. In effetti questa soluzione compila, ma non funziona e solleva una **ClassCastException**:

```
//: generics/ArrayOfGeneric.java

public class ArrayOfGeneric {
  static final int SIZE = 100;
  static Generic<Integer>[] gia;
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    // Compila; produce una ClassCastException:
    //! gia = (Generic<Integer>[])new Object[SIZE];
    // Il tipo a runtime è quello grezzo (cancellato):
    gia = (Generic<Integer>[])new Generic[SIZE];
    System.out.println(gia.getClass().getSimpleName());
    gia[0] = new Generic<Integer>();
    //! gia[1] = new Object(); // Errore di compilazione
    // In fase di compilazione rileva incompatibilita' tra
    // i tipi:
    //! gia[2] = new Generic<Double>();
  }
} /* Output:
Generic[]
*///:~
```

Il problema è che gli array tengono traccia del loro tipo effettivo, stabilito durante la creazione dell'array. Quindi, anche se **gia** è stato sottoposto a cast a **Generic<Integer>[]** tale informazione è valida soltanto al momento della compilazione, e senza l'annotazione **@SuppressWarnings** avreste ottenuto un avvertimento in seguito al cast. In fase di esecuzione si tratta sempre di un array di **Object**, ed è questa la causa del problema. L'unico modo per generare con successo un array di tipo generico consiste nel creare un nuovo array del tipo cancellato e sottoporlo a cast.

Esaminate questo esempio, leggermente più sofisticato. Considerate un semplice wrapper generico che ingloba un array:

```
//: generics/GenericArray.java

public class GenericArray<T> {
  private T[] array;
  @SuppressWarnings("unchecked")
  public GenericArray(int sz) {
    array = (T[])new Object[sz];
  }
  public void put(int index, T item) {
    array[index] = item;
  }
  public T get(int index) { return array[index]; }
  // Metodo che espone la rappresentazione sottostante:
  public T[] rep() { return array; }
  public static void main(String[] args) {
    GenericArray<Integer> gai =
      new GenericArray<Integer>(10);
    // Questo provoca una ClassCastException:
    //! Integer[] ia = gai.rep();
    // Questo e' OK:
    Object[] oa = gai.rep();
  }
} ///:~
```

Come in precedenza, non è possibile affermare che l'array **T[]** è uguale a **new T[sz]**, pertanto viene creato un array di oggetti che sono poi sottoposti a cast.

Il metodo **rep()** restituisce un **T[]**, che in **main()** dovrebbe essere un **Integer[]** per **gai**; se lo chiamate e provate a gestire il risultato come riferimento a **Integer[]** otterrete però una **ClassCastException**, sempre perché **Object[]** è il tipo effettivo a runtime.

Compilando **GenericArray.java** dopo avere commentato l'annotazione @ **SuppressWarnings**, il compilatore visualizzerà un avvertimento:

```
Note: GenericArray.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

In questo caso specifico si è ottenuto un solo avvertimento (che si presume relativo al casting), ma per assicurarvene dovete compilare il codice con l'opzione **-Xlint:unchecked**:

```
GenericArray.java:7: warning: [unchecked] unchecked cast
found   : java.lang.Object[]
required: T[]
    array = (T[])new Object[sz];
                 ^
1 warning
```

È evidente che il compilatore non ha "gradito" questo cast. Poiché questi avvertimenti potrebbero diventare fastidiosi, la cosa migliore che potreste fare è disattivarli con **@SuppressWarnings**: in questo modo, quando vi verrà visualizzato un avvertimento saprete che varrà effettivamente la pena di indagare sul problema.

Tenuto conto della cancellazione, in fase di esecuzione il tipo di array può essere soltanto **Object[]**; convertendolo immediatamente in **T[]** mediante un cast, al momento della compilazione il tipo di array effettivo verrà perso e il compilatore potrebbe così non eseguire alcuni controlli di errore. Per tale motivo, è preferibile servirsi di un **Object[]** all'interno della **Collection** e aggiungere un cast a **T** quando si utilizza un elemento di array. Osservate come questo meccanismo si applica all'esempio **GenericArray.java**:

```
//: generics/GenericArray2.java

public class GenericArray2<T> {
  private Object[] array;
  public GenericArray2(int sz) {
    array = new Object[sz];
  }
  public void put(int index, T item) {
    array[index] = item;
  }
  @SuppressWarnings("unchecked")
  public T get(int index) { return (T)array[index]; }
  @SuppressWarnings("unchecked")
  public T[] rep() {
    return (T[])array; // Avvertimento: unchecked cast
```

```
  }
  public static void main(String[] args) {
    GenericArray2<Integer> gai =
      new GenericArray2<Integer>(10);
    for(int i = 0; i < 10; i ++)
      gai.put(i, i);
    for(int i = 0; i < 10; i ++)
      System.out.print(gai.get(i) + " ");
    System.out.println();
    try {
      Integer[] ia = gai.rep();
    } catch(Exception e) { System.out.println(e); }
  }
} /* Output: (Esempio)
0 1 2 3 4 5 6 7 8 9
java.lang.ClassCastException: [Ljava.lang.Object; cannot be
cast to [Ljava.lang.Integer;
*///:~
```

A prima vista l'esempio non sembra molto diverso: è stato semplicemente spostato il casting e, senza l'annotazione **@SuppressWarnings**, otterreste alcuni avvertimenti "unchecked". Tuttavia la rappresentazione interna ora è **Object[]**, non **T[]**.

Quando chiamate **get()** viene eseguito il casting dell'oggetto a **T**, che è effettivamente il tipo corretto, pertanto il casting è un'operazione sicura. Se chiamate il metodo **rep()**, però, questo tenterà ancora di eseguire la conversione da **Object[]** a **T[]**, che essendo errata produrrà un avvertimento in fase di compilazione e un'eccezione a runtime: non vi è quindi modo di sovvertire il tipo dell'array sottostante, che può essere soltanto **Object[]**.

Il vantaggio di considerare internamente **array** come **Object[]** anziché come **T[]** è che in questo modo è meno probabile dimenticare il tipo di runtime dell'array e introdurre un bug, nonostante la maggior parte (e forse la totalità) di questi errori venga rapidamente identificata in fase di esecuzione.

Per il nuovo codice dovreste passare un contrassegno (*token*) di tipo. In questo caso, **GenericArray** diventa:

```
//: generics/GenericArrayWithTypeToken.java
import java.lang.reflect.*;
```

```
public class GenericArrayWithTypeToken<T> {
  private T[] array;
  @SuppressWarnings("unchecked")
  public GenericArrayWithTypeToken(Class<T> type, int sz) {
    array = (T[])Array.newInstance(type, sz);
  }
  public void put(int index, T item) {
    array[index] = item;
  }
  public T get(int index) { return array[index]; }
  // Espone la rappresentazione sottostante:
  public T[] rep() { return array; }
  public static void main(String[] args) {
    GenericArrayWithTypeToken<Integer> gai =
      new GenericArrayWithTypeToken<Integer>(
        Integer.class, 10);
    // Ora questo funziona:
    Integer[] ia = gai.rep();
  }
} ///:~
```

Il token di tipo **Class<T>** viene passato nel costruttore per il ripristino dalla cancellazione, consentendovi così di generare il tipo reale dell'array di cui avete bisogno, anche se l'avvertimento prodotto dal casting deve essere soppresso con **@SuppressWarnings**. Una volta ottenuto il tipo reale potrete restituirlo e ottenere i risultati desiderati, come vedete in **main()**: il tipo a runtime di **array** è il tipo esatto **T[]**.

Purtroppo, se osservate il codice sorgente delle librerie standard di Java SE5 noterete numerose ricorrenze di cast da array di **Object** a tipi parametrizzati. Per esempio, questo è uno dei costruttori di **ArrayList**, adeguatamente ripulito e semplificato:

```
public ArrayList(Collection c) {
  size = c.size();
  elementData = (E[])new Object[size];
  c.toArray(elementData);
}
```

Se analizzate **ArrayList.java** vedrete numerosi cast di questo tipo. Che cosa accade durante la compilazione?

```
Note: ArrayList.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Di certo le librerie standard generano numerosi avvertimenti in fase di compilazione. Se avete lavorato con il linguaggio C, soprattutto con versioni pre-ANSI, ricorderete un effetto particolare degli avvertimenti: quando scoprite di poterli ignorare, non esitate a farlo. Per questa ragione è meglio che non venga prodotto alcun messaggio in fase di compilazione, a meno che il programmatore non debba intervenire in qualche modo.

Neal Gafter, uno dei direttori dello sviluppo di Java SE5, nel suo weblog (*http://gafter.blogspot.com/2004/09/puzzling-through-erasure-answer.html*) fa rilevare la sua pigrizia durante la riscrittura delle librerie Java, e soprattutto sottolinea che il suo esempio non dovrebbe essere imitato. Neal precisa però che non avrebbe potuto mettere mano al codice di libreria Java senza alterare l'interfaccia esistente. Quindi, sebbene nel codice sorgente delle librerie Java figurino determinate forme idiomatiche, non significa che debbano essere considerate l'approccio corretto: ricordate che quando esaminate il codice sorgente delle librerie non potete dare per scontato che si tratti di un esempio utilizzabile nel vostro codice.

## Limiti

Nel corso di questo capitolo (si veda il paragrafo "Approccio di C++") si è già accennato al concetto di limiti (*bound*), i quali permettono di imporre vincoli sui tipi di parametro utilizzabili con i generici. Anche se grazie ai limiti è possibile applicare regole relative ai tipi cui possono essere applicati i generici, un effetto potenzialmente più importante è che offrono la possibilità di chiamare metodi i quali rientrano nei tipi sottoposti a vincolo.

Poiché la cancellazione rimuove le informazioni sul tipo, i soli metodi richiamabili da un parametro generico non limitato sono quelli disponibili per **Object**. Tuttavia, se siete in grado di limitare quel parametro a un sottoinsieme di tipi potrete chiamare i metodi presenti in quel sottoinsieme.

Per impostare questo vincolo, i generici Java riutilizzano la parola chiave **extends**. È essenziale comprendere che, nel contesto dei limiti generici, **extends** assume un significato molto diverso da quello standard.

Questo esempio illustra i concetti fondamentali dei limiti:

```
//: generics/BasicBounds.java

interface HasColor { java.awt.Color getColor(); }

class Colored<T extends HasColor> {
  T item;
  Colored(T item) { this.item = item; }
  T getItem() { return item; }
  // Il limite consente di chiamare un metodo:
  java.awt.Color color() { return item.getColor(); }
}

class Dimension { public int x, y, z; }

// Questo non funziona: prima la classe, poi le interfacce:
// class ColoredDimension<T extends HasColor & Dimension> {

// Limiti multipli:
class ColoredDimension<T extends Dimension & HasColor> {
  T item;
  ColoredDimension(T item) { this.item = item; }
  T getItem() { return item; }
  java.awt.Color color() { return item.getColor(); }
  int getX() { return item.x; }
  int getY() { return item.y; }
  int getZ() { return item.z; }
}

interface Weight { int weight(); }

// Come nel caso dell'ereditarieta', potete avere una sola
// classe concreta ma interfacce multiple
class Solid<T extends Dimension & HasColor & Weight> {
  T item;
  Solid(T item) { this.item = item; }
  T getItem() { return item; }
  java.awt.Color color() { return item.getColor(); }
```

```
  int getX() { return item.x; }
  int getY() { return item.y; }
  int getZ() { return item.z; }
  int weight() { return item.weight(); }
}

class Bounded
extends Dimension implements HasColor, Weight {
  public java.awt.Color getColor() { return null; }
  public int weight() { return 0; }
}

public class BasicBounds {
  public static void main(String[] args) {
    Solid<Bounded> solid =
      new Solid<Bounded>(new Bounded());
    solid.color();
    solid.getY();
    solid.weight();
  }
} ///:~
```

Potreste osservare che **BasicBounds.java** sembra contenere ridondanze che potrebbero essere eliminate per mezzo dell'ereditarietà. Di seguito, notate come ogni livello di ereditarietà aggiunga vincoli ai limiti:

```
//: generics/InheritBounds.java

class HoldItem<T> {
  T item;
  HoldItem(T item) { this.item = item; }
  T getItem() { return item; }
}

class Colored2<T extends HasColor> extends HoldItem<T> {
  Colored2(T item) { super(item); }
  java.awt.Color color() { return item.getColor(); }
}
```

```
class ColoredDimension2<T extends Dimension & HasColor>
extends Colored2<T> {
  ColoredDimension2(T item) {  super(item); }
  int getX() { return item.x; }
  int getY() { return item.y; }
  int getZ() { return item.z; }
}

class Solid2<T extends Dimension & HasColor & Weight>
extends ColoredDimension2<T> {
  Solid2(T item) {  super(item); }
  int weight() { return item.weight(); }
}

public class InheritBounds {
  public static void main(String[] args) {
    Solid2<Bounded> solid2 =
      new Solid2<Bounded>(new Bounded());
    solid2.color();
    solid2.getY();
    solid2.weight();
  }
} ///:~
```

**HoldItem** contiene semplicemente un oggetto, quindi questo comportamento viene ereditato da **Colored2**, il quale richiede inoltre che il parametro sia conforme a **HasColor**. **ColoredDimension2** e **Solid2** estendono ulteriormente la gerarchia e aggiungono i limiti a ogni livello. Ora i metodi vengono ereditati, di conseguenza non devono essere ripetuti in ogni classe.

Ecco un esempio con più livelli:

```
//: generics/EpicBattle.java
// Dimostra i limiti nei generici Java.
import java.util.*;

interface SuperPower {}
interface XRayVision extends SuperPower {
```

```
  void seeThroughWalls();
}
interface SuperHearing extends SuperPower {
  void hearSubtleNoises();
}
interface SuperSmell extends SuperPower {
  void trackBySmell();
}

class SuperHero<POWER extends SuperPower> {
  POWER power;
  SuperHero(POWER power) { this.power = power; }
  POWER getPower() { return power; }
}

class SuperSleuth<POWER extends XRayVision>
extends SuperHero<POWER> {
  SuperSleuth(POWER power) { super(power); }
  void see() { power.seeThroughWalls(); }
}

class CanineHero<POWER extends SuperHearing & SuperSmell>
extends SuperHero<POWER> {
  CanineHero(POWER power) { super(power); }
  void hear() { power.hearSubtleNoises(); }
  void smell() { power.trackBySmell(); }
}

class SuperHearSmell implements SuperHearing, SuperSmell {
  public void hearSubtleNoises() {}
  public void trackBySmell() {}
}

class DogBoy extends CanineHero<SuperHearSmell> {
  DogBoy() { super(new SuperHearSmell()); }
}
```

```
public class EpicBattle {
  // Limiti in metodi generici:
  static <POWER extends SuperHearing>
  void useSuperHearing(SuperHero<POWER> hero) {
    hero.getPower().hearSubtleNoises();
  }
  static <POWER extends SuperHearing & SuperSmell>
  void superFind(SuperHero<POWER> hero) {
    hero.getPower().hearSubtleNoises();
    hero.getPower().trackBySmell();
}
  public static void main(String[] args) {
    DogBoy dogBoy = new DogBoy();
    useSuperHearing(dogBoy);
    superFind(dogBoy);
    // Potete fare questo:
    List<? extends SuperHearing> audioBoys;
    // Ma non questo:
    // List<? extends SuperHearing & SuperSmell> dogBoys;
  }
} ///:~
```

Notate che i metacaratteri ? (che vedrete nel prossimo paragrafo) sono vincolati a un solo limite.

> **Esercizio 25** (2) Create due interfacce e una classe che le implementi entrambe. Generate poi due metodi generici, uno il cui parametro dell'argomento sia limitato dalla prima interfaccia e uno il cui parametro dell'argomento sia limitato dalla seconda interfaccia. Create un'istanza della classe che implementi entrambe le interfacce e dimostrate che può essere utilizzata con entrambi i metodi generici.

## Metacaratteri

Nel Volume 1, Capitolo 11 e nel Capitolo 2 di questo volume, avete già visto alcuni semplici esempi di utilizzo dei *metacaratteri*, in particolare quello dei punti interrogativi nelle espressioni dei generici. In questo paragrafo avrete

modo di esaminarli in modo più approfondito, iniziando con un esempio che mostra un comportamento particolare degli array.

Potete assegnare un array di un tipo derivato a un riferimento a un array del tipo di base:

```
//: generics/CovariantArrays.java

class Fruit {}
class Apple extends Fruit {}
class Jonathan extends Apple {}
class Orange extends Fruit {}

public class CovariantArrays {
  public static void main(String[] args) {
    Fruit[] fruit = new Apple[10];
    fruit[0] = new Apple(); // OK
    fruit[1] = new Jonathan(); // OK
    // Il tipo a runtime e' Apple[], non Fruit[] o Orange[]:
  try {
      // Il compilatore vi permette di aggiungere Fruit:
      fruit[0] = new Fruit(); // ArrayStoreException
    } catch(Exception e) { System.out.println(e); }
    try {
      // Il compilatore vi permette di aggiungere Orange:
      fruit[0] = new Orange(); // ArrayStoreException
    } catch(Exception e) { System.out.println(e); }
  }
} /* Output:
java.lang.ArrayStoreException: Fruit
java.lang.ArrayStoreException: Orange
*///:~
```

La prima riga in **main()** genera un array di **Apple** e lo assegna a un riferimento a un array di **Fruit**. Questo ha perfettamente senso dal momento che una **Apple** è un tipo di **Fruit**, quindi un array di **Apple** dovrebbe anche essere un array di **Fruit**.

Tuttavia, se il tipo di array effettivo è **Apple[]** dovreste essere soltanto in grado di inserire nell'array un'**Apple** o un suo sottotipo, operazione che infatti funziona sia in fase di compilazione sia a runtime. Notate però che il compilatore vi permette di inserire nell'array un oggetto **Fruit**. Questo ha senso per il compilatore, che dispone di un riferimento **Fruit[]**: perché non permettere a un oggetto **Fruit** o qualsiasi oggetto che derivi da esso, come **Orange**, di essere inserito nell'array? Pertanto, in fase di compilazione tale operazione è consentita. Il meccanismo di runtime, tuttavia, sa che sta gestendo un array **Apple[]** e solleva un'eccezione quando nell'array viene incluso un tipo diverso.

In questo caso non si tratta realmente di un "upcast", bensì dell'assegnazione di un array a un altro. Per sua natura l'array contiene altri oggetti, ma poiché esiste la capacità di eseguire l'upcast, è chiaro che gli oggetti array mantengono le regole sul tipo di oggetti che contengono. È come se gli array fossero "consapevoli" di quanto devono contenere, in modo che tra i controlli in fase di compilazione e quelli a runtime non sia possibile abusare di loro.

Questa gestione degli array non è poi così terribile, poiché vi consente di scoprire in fase di esecuzione di avere inserito un tipo sbagliato. Tuttavia, uno degli obiettivi principali dei generici è quello di spostare l'individuazione di questi errori nella fase di compilazione: che cosa accade, quindi, utilizzando i contenitori generici invece degli array?

```
//: generics/NonCovariantGenerics.java
// {CompileTimeError} (Non compilera')
import java.util.*;

public class NonCovariantGenerics {
  // Errore di compilazione: incompatible types:
  List<Fruit> flist = new ArrayList<Apple>();
} ///:~
```

Benché a prima vista possiate interpretare questo comportamento come l'impossibilità di assegnare un contenitore di **Apple** a un contenitore di **Fruit**, ricordate che i generici non sono soltanto limitati ai contenitori. In effetti, il codice di questo esempio dovrebbe essere interpretato come: "Non è possibile assegnare un generico *che implica* **Apple** a un generico *che implica* **Fruit**". Se il compilatore, come nel caso degli array, ottiene sufficienti informazioni dal codice per determinare quali contenitori sono coinvolti, è possibile che offra una certa flessibilità.

Tuttavia in questo caso specifico il compilatore non possiede queste informazioni, quindi rifiuta l'"upcast". In realtà questo non è comunque un "upcast", poiché una **List** di **Apple** non è una **List** di **Fruit**. Una **List** di **Apple** conterrà **Apple** e sottotipi di **Apple**, mentre in una **List** di **Fruit** potrà esserci qualsiasi genere di **Fruit**; incluse le **Apple**, certamente, ma senza che questo ne faccia una **List** di **Apple**: rimarrà una **List** di **Fruit**. Una **List** di **Apple** non è quindi equivalente, a livello di tipo, a una **List** di **Fruit**, anche se un'**Apple** è un tipo di **Fruit**.

Il vero problema è che si sta parlando del *tipo del contenitore*, non del tipo che il contenitore *deve contenere*. A differenza degli array, i generici non dispongono della funzionalità di *covarianza* nativa, perché gli array sono interamente definiti a livello di linguaggio e possono così beneficiare di controlli in fase di compilazione e di esecuzione; con i generici, invece, il compilatore e il motore di runtime non possono sapere che cosa volete fare con i vostri tipi, né quali regole dovrebbero essere applicate.

Talvolta, però, vorreste poter impostare qualche tipo di relazione di upcasting tra i due elementi. Questo è esattamente lo scopo dei metacaratteri.

```
//: generics/GenericsAndCovariance.java
import java.util.*;

public class GenericsAndCovariance {
  public static void main(String[] args) {
    // I metacaratteri consentono la covarianza:
    List<? extends Fruit> flist = new ArrayList<Apple>();
    // Errore di compilazione: can't add any type of object:
    // flist.add(new Apple());
    // flist.add(new Fruit());
    // flist.add(new Object());
    flist.add(null); // Possibile, ma poco interessante
    // Sapete che restituisce almeno Fruit:
    Fruit f = flist.get(0);
  }
} ///:~
```

Il tipo di **flist** ora è **List<? extends Fruit>**, che potete interpretare come "una **List** di qualsiasi tipo derivato da **Fruit**". Questo non significa tuttavia che **List** conterrà qualsiasi tipo di **Fruit**. Il metacarattere fa riferimento a un tipo

definito, quindi significa: "un tipo specifico che il riferimento **flist** non menziona". Cosicché la **List** cui è assegnato deve contenere un tipo specificato come **Fruit** o **Apple**, ma ai fini dell'upcast a **flist** questo tipo è un "tipo qualsiasi".

Se l'unico vincolo è che **List** contenga **Fruit** o un suo sottotipo, ma in realtà a voi non importa quale sia, che cosa potete fare con questa **List**? Se non sapete quale tipo **List** può contenere, come potete aggiungere con sicurezza un oggetto? Come avete già visto per l'"upcast" sull'array in **CovariantArrays.java** anche questo non è possibile, con l'unica differenza che sarà il compilatore (e non il sistema a runtime) a impedirvi di aggiungere un tipo. Il problema sarà analizzato tra breve.

Potreste ribattere che le cose sono in qualche modo sfuggite di mano, perché ora non è più possibile aggiungere nemmeno una **Apple** a una **List** che avete appena definito come contenitore di **Apple**. Avete ragione, ma il problema è che il compilatore non lo sa: dal suo punto di vista **List<? extends Fruit>** potrebbe perfettamente puntare a una **List<Orange>**. Una volta eseguito questo tipo di "upcast" perdete la capacità di passare qualsiasi cosa, anche un **Object**.

D'altra parte, la chiamata di un metodo che restituisce **Fruit** è un comportamento sicuro: il compilatore la permette perché sa che qualsiasi cosa sia in **List** deve essere almeno di tipo **Fruit**.

> **Esercizio 26** (2) Dimostrate la covarianza degli array utilizzando le classi **Number** e **Integer**.
>
> **Esercizio 27** (2) Utilizzando le classi **Number** e **Integer** dimostrate che la covarianza non funziona con le **List**, quindi introducete l'utilizzo dei metacaratteri.

## *Quanto è "intelligente" il compilatore?*

Ora potreste pensare che vi sia impossibile chiamare qualunque metodo accetti degli argomenti, ma considerate questo esempio:

```
//: generics/CompilerIntelligence.java
import java.util.*;

public class CompilerIntelligence {
  public static void main(String[] args) {
    List<? extends Fruit> flist =
      Arrays.asList(new Apple());
```

```
    Apple a = (Apple)flist.get(0); // Nessun avvertimento
    flist.contains(new Apple()); // L'argomento e' 'Object'
    flist.indexOf(new Apple()); // L'argomento e' 'Object'
  }
} ///:~
```

Notate che le chiamate a **contains()** e **indexOf()** che accettano oggetti **Apple** come argomenti vanno bene; questo potrebbe voler dire che il compilatore analizza il codice per verificare se un metodo modifica il suo oggetto.

Consultando la documentazione per **ArrayList**, rileverete che il compilatore non è poi così astuto. Mentre **add()** accetta come argomento un parametro generico di tipo, **contains()** e **indexOf()** richiedono argomenti di tipo **Object**; pertanto, quando specificate **ArrayList<? extends Fruit >** l'argomento di **add()** diventa **'? extends Fruit'**. Da questa descrizione il compilatore non potrà sapere quale specifico sottotipo di **Fruit** è richiesto in questo caso, di conseguenza non accetterà alcun tipo di **Fruit**. Non ha importanza se prima eseguite un upcast di **Apple** a **Fruit**, perché se nell'elenco degli argomenti è presente un metacarattere il compilatore si rifiuterà comunque di chiamare un metodo come **add()**.

Con i metodi **contains()** e **indexOf()** gli argomenti sono di tipo **Object**, che non implicano metacaratteri; di conseguenza il compilatore accetta la chiamata. È quindi responsabilità del progettista della classe generica decidere quali chiamate siano "adatte" e utilizzare i tipi **Object** come loro argomenti. Per impedire una chiamata quando il tipo utilizzato contiene metacaratteri, precisate il parametro di tipo nell'elenco degli argomenti.

Osservate questa tecnica in una classe **Holder** molto semplice:

```
//: generics/Holder.java

public class Holder<T> {
  private T value;
  public Holder() {}
  public Holder(T val) { value = val; }
  public void set(T val) { value = val; }
  public T get() { return value; }
  public boolean equals(Object obj) {
    return value.equals(obj);
  }
```

```
  public static void main(String[] args) {
    Holder<Apple> Apple = new Holder<Apple>(new Apple());
    Apple d = Apple.get();
    Apple.set(d);
    // Holder<Fruit> Fruit = Apple; // Non ammette l'upcast
    Holder<? extends Fruit> fruit = Apple; // OK
    Fruit p = fruit.get();
    d = (Apple)fruit.get(); // Restituisce 'Object'
    try {
      Orange c = (Orange)fruit.get(); // Nessun avvertimento
    } catch(Exception e) { System.out.println(e); }
    // fruit.set(new Apple()); // Impossibile chiamare set()
    // fruit.set(new Fruit()); // Impossibile chiamare set()
    System.out.println(fruit.equals(d)); // OK
  }
} /* Output: (Esempio)
java.lang.ClassCastException: Apple cannot be cast to
Orange
true
*///:~
```

**Holder** dispone di un metodo **set()** che accetta una **T**, un **get()** che restituisce una **T** e un **equals()** cha accetta un **Object**. Come avete già visto, se create **Holder<Apple>** non potete eseguirne l'upcast a **Holder<Fruit>**, ma potete farlo a **Holder<? extends fruit>**. Se chiamate **get()** il metodo restituirà soltanto un **Fruit**, vale a dire tutto ciò che sa, tenuto conto del limite "qualsiasi cosa estenda **Fruit**". Nel caso abbiate qualche informazione aggiuntiva potreste eseguire il cast a uno specifico tipo di **Fruit**, e non ottenendo alcun avvertimento da parte del compilatore, rischiereste di sollevare una **ClassCastException**. Neppure il metodo **set()** funzionerà né con **Apple** né con **Fruit** perché anche l'argomento del metodo **set()** è "**? extends Fruit**" ("qualsiasi cosa"), e come tale non potrà essere verificato dal compilatore per quanto riguarda la sicurezza dei tipi per "qualsiasi cosa".

Tuttavia il metodo **equals()** funziona benissimo, perché accetta un **Object** anziché una **T** come parametro: ne deriva che il compilatore si preoccupa soltanto dei tipi di oggetti passati e restituiti, senza analizzare il codice alla ricerca di eventuali effettive operazioni di lettura o scrittura.

## *Controvarianza*

È possibile anche procedere in senso inverso utilizzando i cosiddetti "metacaratteri di supertipo" o sovratipo (*supertype wildcards*). In questo caso si tratta di affermare che il metacarattere è limitato da qualunque classe di base di una determinata classe, specificando **<? super MyClass>** o anche utilizzando un parametro di tipo, come in **<? super T>**; ricordate che non è permesso assegnare un limite di supertipo a un parametro generico, cioè non potete scrivere **<T super MyClass>**. Questa tecnica vi consente in assoluta sicurezza di fare passare un oggetto tipizzato in un oggetto generico. Per esempio, con i metacaratteri di supertipo potete scrivere in una **Collection**:

```
//: generics/SuperTypeWildcards.java
import java.util.*;

public class SuperTypeWildcards {
  static void writeTo(List<? super Apple> apples) {
    apples.add(new Apple());
    apples.add(new Jonathan());
    // apples.add(new Fruit()); // Errore
  }
} ///:~
```

L'argomento **apples** è una **List** qualsiasi che costituisce il tipo di base di **Apple**; sapete così che è sicuro aggiungere un'**Apple** o un sottotipo di **Apple**. Poiché il *limite inferiore* è **Apple**, tuttavia, non siete certi che sia sicuro aggiungere **Fruit** a questa **List**, perché in tal caso si potrebbero aggiungere alla **List** anche oggetti di tipo non **Apple**, che violerebbero la sicurezza di tipo statica.

Potete quindi iniziare a considerare i limiti di sottotipo e supertipo in termini di "come scrivere (cioè passare in un metodo) in un tipo generico" e "come leggere (cioè restituire da un metodo) da un tipo generico", rispettivamente. I limiti di supertipo allentano i vincoli su ciò che è possibile passare a un metodo:

```
//: generics/GenericWriting.java
import java.util.*;

public class GenericWriting {
  static <T> void writeExact(List<T> list, T item) {
    list.add(item);
  }
```

```
  static List<Apple> apples = new ArrayList<Apple>();
  static List<Fruit> fruit = new ArrayList<Fruit>();
  static void f1() {
    writeExact(apples, new Apple());
    // writeExact(fruit, new Apple()); // Errore:
    // Incompatible types: found Fruit, required Apple
  }
  static <T> void
  writeWithWildcard(List<? super T> list, T item) {
    list.add(item);
  }
  static void f2() {
    writeWithWildcard(apples, new Apple());
    writeWithWildcard(fruit, new Apple());
  }
  public static void main(String[] args) { f1(); f2(); }
} ///:
```

Il metodo **writeExact()** utilizza un tipo di parametro esatto, cioè privo di metacaratteri. Potete vedere che questo funziona perfettamente in **f1()**, purché in **List<Apple>** vengano inserite soltanto **Apple**. Tuttavia **writeExact()** non vi permette di inserire una **Apple** in una **List<Fruit>**, anche se sapete che ciò è tecnicamente possibile.

Nel metodo **writeWithWildcard()** ora l'argomento è **List<? super T>**, pertanto **List** contiene un tipo specifico derivato da **T**. È quindi sicuro passare una **T** o un suo derivato come argomento ai metodi di **List**; questo risulta evidente in **f2()**, dove oltre a poter inserire un'**Apple** in **List<Apple>**, come in precedenza, ora è anche possibile inserire un'**Apple** in **List<Fruit>**, come previsto.

Potete eseguire questo stesso tipo di analisi come revisione della covarianza e dei metacaratteri:

```
//: generics/GenericReading.java
import java.util.*;

public class GenericReading {
  static <T> T readExact(List<T> list) {
    return list.get(0);
  }
```

```
  static List<Apple> apples = Arrays.asList(new Apple());
  static List<Fruit> fruit = Arrays.asList(new Fruit());
  // Un metodo static si adatta a ogni chiamata:
  static void f1() {
    Apple a = readExact(apples);
    Fruit f = readExact(fruit);
    f = readExact(apples);
  }
  // Tuttavia, in caso di classe, il tipo di quest'ultima
  // viene impostato al momento dell'istanziazione della
  // classe:
static class Reader<T> {
    T readExact(List<T> list) { return list.get(0); }
  }
  static void f2() {
    Reader<Fruit> fruitReader = new Reader<Fruit>();
    Fruit f = fruitReader.readExact(fruit);
    // Fruit a = fruitReader.readExact(apples); // Errore:
    // readExact(List<Fruit>) cannot be
    // applied to (List<Apple>).
  }
  static class CovariantReader<T> {
    T readCovariant(List<? extends T> list) {
      return list.get(0);
    }
  }
  static void f3() {
    CovariantReader<Fruit> fruitReader =
      new CovariantReader<Fruit>();
    Fruit f = fruitReader.readCovariant(fruit);
    Fruit a = fruitReader.readCovariant(apples);
  }
  public static void main(String[] args) {
    f1(); f2(); f3();
  }
} ///:~
```

Come in precedenza, il primo metodo **readExact()** utilizza il tipo esatto. Quindi, quando vi servite del tipo esatto senza i metacaratteri potete scrivere e leggere quel tipo esatto in una **List**. Inoltre, per il valore di ritorno il metodo generico **static readExact()** "si adatta" convenientemente a ogni chiamata di metodo e restituisce un'**Apple** da **List<Apple>** e un **Fruit** da **List<Fruit>**, come risulta evidente in **f1()**. Pertanto, se riuscite a cavarvela con un metodo generico **static**, è indubbio che non avrete bisogno della covarianza soltanto per la lettura.

In caso di classe generica, tuttavia, il parametro viene impostato per la classe nel momento in cui la si istanzia. Come potete vedere in **f2()** l'istanza di **fruitReader** può leggere un elemento di **Fruit** da **List<Fruit>** dal momento che è il suo tipo esatto. Ma una **List<Apple>** dovrebbe anche produrre oggetti **Fruit**, e ciò non è permesso da **fruitReader**.

Per risolvere il problema, il metodo **CovariantReader.readCovariant()** accetta una **List<? Extends T>** rendendo perciò sicura la lettura di una **T** da questa lista, tenuto conto che sapete che essa contiene o una **T** o qualcosa derivato da **T**. Ecco quindi che in **f3()** è finalmente possibile leggere un **Fruit** da **List<Apple>**.

> **Esercizio 28** (4) Create una classe generica **Generic1<T>** con un solo metodo che *accetta* un argomento di tipo **T**, poi una seconda classe generica **Generic2<T>** con un metodo che *restituisce* un argomento di tipo **T**. Scrivete poi un metodo generico con un argomento *controvariante* rispetto alla prima classe generica che chiama il metodo di **Generic1**, e un secondo metodo generico con un argomento *covariante* rispetto alla seconda classe generica che chiama il metodo di **Generic2**. Testate questo schema utilizzando la libreria **typeinfo.pets**.

## *Metacaratteri senza limiti*

Il metacarattere senza limiti **<?>** (*unbounded wildcard*) sembra significare "qualsiasi cosa", di conseguenza questa sintassi dovrebbe equivalere all'utilizzo di un tipo grezzo. Effettivamente, a prima vista il compilatore sembra condividere questa tesi:

```
//: generics/UnboundedWildcards1.java
import java.util.*;

public class UnboundedWildcards1 {
  static List list1;
```

```
  static List<?> list2;
  static List<? extends Object> list3;
  static void assign1(List list) {
    list1 = list;
    list2 = list;
    // list3 = list; // Avvertimento: unchecked conversion
    // Found: List, Required: List<? extends Object>
  }
  static void assign2(List<?> list) {
    list1 = list;
    list2 = list;
    list3 = list;
  }
  static void assign3(List<? extends Object> list) {
    list1 = list;
    list2 = list;
    list3 = list;
  }
  public static void main(String[] args) {
    assign1(new ArrayList());
    assign2(new ArrayList());
    // assign3(new ArrayList()); // Avvertimento:
    // Unchecked conversion. Found: ArrayList
    // Required: List<? extends Object>
    assign1(new ArrayList<String>());
    assign2(new ArrayList<String>());
    assign3(new ArrayList<String>());
    // Entrambe le forme sono accettabili come List<?>:

    List<?> wildList = new ArrayList();
    wildList = new ArrayList<String>();
    assign1(wildList);
    assign2(wildList);
    assign3(wildList);
  }
} ///:~
```

In molti casi analoghi a quelli qui presentati, il compilatore potrebbe essere meno "attento" al fatto che utilizziate un tipo grezzo o **<?>**. In queste situazioni **<?>** può essere interpretato come accessorio ornamentale, mentre è essenziale perché afferma che il codice è stato scritto in un'ottica di generici Java, e che nel caso in questione non si vuole utilizzare un tipo grezzo, bensì specificare che il parametro generico può contenere qualsiasi tipo.

Un secondo esempio illustra un impiego di grande interesse dei metacaratteri senza limiti. Quando dovete gestire parametri generici multipli, può essere importante fare in modo che un parametro sia di qualsiasi tipo mentre si stabilisce un tipo particolare per gli altri:

```
//: generics/UnboundedWildcards2.java
import java.util.*;

public class UnboundedWildcards2 {
  static Map map1;
  static Map<?,?> map2;
  static Map<String,?> map3;
  static void assign1(Map map) { map1 = map; }
  static void assign2(Map<?,?> map) { map2 = map; }
  static void assign3(Map<String,?> map) { map3 = map; }
  public static void main(String[] args) {
    assign1(new HashMap());
    assign2(new HashMap());
    // assign3(new HashMap()); // Avvertimento:
    // Unchecked conversion. Found: HashMap
    // Required: Map<String,?>
    assign1(new HashMap<String,Integer>());
    assign2(new HashMap<String,Integer>());
    assign3(new HashMap<String,Integer>());
  }
} ///:~
```

Di nuovo, come vedete in **Map<?,?>**, quando utilizzate soltanto i metacaratteri senza limiti il compilatore non sembra fare distinzione rispetto a una **Map** grezza. Inoltre, **UnboundedWildcards1.java** evidenzia che il compilatore considera **List<?>** e **List<? extends Object>** in modo diverso.

L'aspetto fuorviante è il fatto che il compilatore non sempre tiene conto della differenza esistente, per esempio tra **List** e **List<?>**, cosicché queste due liste potrebbero sembrare identiche. Infatti, poiché un argomento generico cancella fino al suo primo limite, **List<?>** *sembra equivalente* a **List<Object>** e **List** *è effettivamente* anche **List<Object>**: tenete presente, però, che nessuna di queste affermazioni è del tutto vera. In realtà **List** significa "una **List** grezza che contiene qualsiasi tipo **Object**", mentre **List<?>** significa "una **List** non grezza di un *certo tipo specifico*, che tuttavia è ignoto".

Quando il compilatore si interessa realmente alla differenza fra tipi grezzi e tipi che coinvolgono metacaratteri senza limiti? L'esempio seguente si serve della classe precedentemente definita come **Holder<T>**; contiene metodi che accettano **Holder** come argomento ma in forme diverse, ovvero come tipo grezzo, con un parametro di tipo specifico e con un parametro metacarattere senza limite:

```
//: generics/Wildcards.java
// Analisi del significato dei metacaratteri.

public class Wildcards {
   // Argomento grezzo:
  static void rawArgs(Holder holder, Object arg) {
    // holder.set(arg); // Avvertimento:
    //   Unchecked call to set(T) as a
    //   member of the raw type Holder
    // holder.set(new Wildcards()); // Stesso avvertimento

    // Non potete fare questo, perche' non avete 'T':
    // T t = holder.get();

    // OK, ma le informazioni sul tipo vengono perse:
    Object obj = holder.get();
  }
  // Simile a rawArgs(),ma genera errori invece di
  // avvertimenti:
  static void unboundedArg(Holder<?> holder, Object arg) {
    // holder.set(arg); // Errore:
    //   set(capture of ?) in Holder<capture of ?>
    //   cannot be applied to (Object)
    // holder.set(new Wildcards()); // Stesso errore
```

```
    // Non potete fare questo, perche' non avete 'T':
    // T t = holder.get();

    // OK, ma le informazioni sul tipo vengono perse:
    Object obj = holder.get();
  }
  static <T> T exact1(Holder<T> holder) {
    T t = holder.get();
    return t;
  }
  static <T> T exact2(Holder<T> holder, T arg) {
    holder.set(arg);
    T t = holder.get();
    return t;
  }
  static <T>
  T wildSubtype(Holder<? extends T> holder, T arg) {
    // holder.set(arg); // Errore:
    //   set(capture of ? extends T) in
    //   Holder<capture of ? extends T>
    //   cannot be applied to (T)
    T t = holder.get();
    return t;
  }
  static <T>
  void wildSupertype(Holder<? super T> holder, T arg) {
    holder.set(arg);
    // T t = holder.get();  // Errore:
    //   Incompatible types: found Object, required T

    // OK, ma le informazioni sul tipo vengono perse:
    Object obj = holder.get();
  }
  public static void main(String[] args) {
    Holder raw = new Holder<Long>();
    // Oppure:
    raw = new Holder();
```

```
Holder<Long> qualified = new Holder<Long>();
Holder<?> unbounded = new Holder<Long>();
Holder<? extends Long> bounded = new Holder<Long>();
Long lng = 1L;

rawArgs(raw, lng);
rawArgs(qualified, lng);
rawArgs(unbounded, lng);
rawArgs(bounded, lng);

unboundedArg(raw, lng);
unboundedArg(qualified, lng);
unboundedArg(unbounded, lng);
unboundedArg(bounded, lng);

// Object r1 = exact1(raw); // Avvertimenti:
//   Unchecked conversion from Holder to Holder<T>
//   Unchecked method invocation: exact1(Holder<T>)
//   is applied to (Holder)
Long r2 = exact1(qualified);
Object r3 = exact1(unbounded); // Deve restituire Object
Long r4 = exact1(bounded);

// Long r5 = exact2(raw, lng); // Avvertimenti:
//   Unchecked conversion from Holder to Holder<Long>
//   Unchecked method invocation: exact2(Holder<T>,T)
//   is applied to (Holder,Long)
Long r6 = exact2(qualified, lng);
// Long r7 = exact2(unbounded, lng); // Errore:
//   exact2(Holder<T>,T) cannot be applied to
//   (Holder<capture of ?>,Long)
// Long r8 = exact2(bounded, lng); // Errore:
//   exact2(Holder<T>,T) cannot be applied
//   to (Holder<capture of ? extends Long>,Long)

// Long r9 = wildSubtype(raw, lng); // Avvertimenti:
//   Unchecked conversion from Holder
//   to Holder<? extends Long>
```

```
        //   Unchecked method invocation:
        //   wildSubtype(Holder<? extends T>,T) is
        //   applied to (Holder,Long)
        Long r10 = wildSubtype(qualified, lng);
        // OK, ma puo' restituire soltanto Object:
        Object r11 = wildSubtype(unbounded, lng);
        Long r12 = wildSubtype(bounded, lng);

        // wildSupertype(raw, lng); // Avvertimenti:
        //   Unchecked conversion from Holder
        //   to Holder<? super Long>
        //   Unchecked method invocation:
        //   wildSupertype(Holder<? super T>,T)
        //   is applied to (Holder,Long)
        wildSupertype(qualified, lng);
        // wildSupertype(unbounded, lng); // Errore:
        //   wildSupertype(Holder<? super T>,T) cannot be
        //   applied to (Holder<capture of ?>,Long)
        // wildSupertype(bounded, lng); // Error:
        //   wildSupertype(Holder<? super T>,T) cannot be
        //  applied to (Holder<capture of ? extends Long>,Long)
    }
} ///:~
```

In **rawArgs()** si sa che **Holder** è un tipo generico benché espresso come tipo grezzo, pertanto il compilatore sa che passare un **Object** a **set()** è rischioso. Trattandosi di un tipo grezzo, potete passare un oggetto di qualunque tipo a **set()** e questo oggetto subirà un upcast a **Object**. Così ogni volta che avete un tipo grezzo, il controllo del compilatore fallisce. La chiamata a **get()** presenta lo stesso comportamento: non c'è nessuna **T**, quindi il risultato può essere soltanto un **Object**.

Potreste essere tentati di ritenere che un **Holder** grezzo e un **Holder<?>** si equivalgano, tuttavia il metodo **unboundedArg()** vi fa subito notare che sono diversi: evidenzia lo stesso genere di problemi, ma li segnala come errori e non come avvertimenti, perché l'oggetto **Holder** grezzo conterrà una combinazione di tipi qualsiasi mentre in **Holder<?>** è una collezione omogenea di un *certo tipo specifico*, al quale non è quindi possibile passare semplicemente un **Object**.

In **exact1()** e **exact2()** vengono utilizzati i parametri generici esatti, senza metacaratteri; notate che **exact2()** ha limiti diversi da **exact1()**, a causa dell'argomento aggiuntivo.

In **wildSubtype()** i vincoli sul tipo di **Holder** sono allentati, per includere un **Holder** di qualsiasi cosa estenda (**extends**) **T**. Anche in questo caso significa che **T** potrebbe essere **Fruit**, mentre **holder** potrebbe anche essere un **Holder<Apple>**. Per impedire che in un **Holder<Apple>** venga inserita una **Orange**, la chiamata a **set()** o qualsiasi metodo che accetti un argomento del parametro di tipo non è ammessa. Tuttavia sapete sempre che qualsiasi cosa provenga da **Holder<? extends Fruit>** sarà quantomeno un **Fruit**, pertanto **get()** o qualsiasi metodo che restituisce il parametro di tipo è consentito.

I metacaratteri di supertipo sono indicati in **wildSupertype()**, che ha un comportamento opposto rispetto a **wildSubtype()**: **holder** può essere un contenitore che include qualsiasi tipo che sia una classe di base di **T**. Quindi **set()** può accettare una **T**, dal momento che tutto ciò che funziona con un tipo di base funzionerà in modo polimorfico con un tipo derivato, di conseguenza anche un **T**. Comunque una chiamata a **get()** non vi sarà di aiuto, poiché il tipo posseduto da **Holder** può essere qualsiasi supertipo, e l'unico che può essere considerato sicuro è **Object**.

Questo esempio mostra anche i limiti di ciò che è possibile e impossibile fare con un parametro senza limiti in **unbounded()**: non potete utilizzare né **get()** né **set()** con una **T**, perché *non avete* una **T**.

In **main()** potete vedere quali metodi possono accettare, senza errori né avvertimenti, quali tipi di argomento. Per compatibilità in vista della migrazione, **rawArgs()** accetterà tutte le varianti di **Holder** senza segnalare avvertimenti. Allo stesso modo, il metodo **unboundedArg()** accetterà tutti i tipi anche se, come si è detto, li gestirà diversamente all'interno del corpo del metodo.

Se passate un riferimento **Holder** grezzo a un metodo che accetta un tipo generico "esatto" (ossia senza metacaratteri) otterrete un avvertimento, perché l'argomento esatto si aspetterà informazioni che non sono disponibili nel tipo grezzo. Infine, se passate un riferimento senza limiti a **exact1()**, non avrete alcuna informazione che vi consenta di determinare il tipo di ritorno.

Come potete vedere **exact2()** è il metodo con più vincoli, poiché pretende esattamente un **Holder<T>** e un argomento di tipo **T**, in mancanza dei quali genererà errori o avvertimenti. Talvolta questo comportamento è accettabile, ma se doveste ritenerlo eccessivamente limitativo potreste servirvi dei metacaratteri, a seconda che vogliate ottenere i valori di ritorno tipizzati dal

vostro argomento generico (come avete visto in **wildSubtype()**) o passare gli argomenti tipizzati al vostro argomento generico (come in **wildSupertype()**).

Quindi, il vantaggio di utilizzare tipi esatti in luogo dei tipi con metacaratteri consiste in un utilizzo più efficace dei parametri generici; l'adozione dei metacaratteri, invece, vi consente di accettare una gamma più vasta di tipi parametrizzati come argomenti. Come vedete, si tratta di accettare il compromesso più adatto alle necessità, caso per caso.

## *Conversione dell'intercettazione*

Una situazione, in particolare, *richiede* l'utilizzo di **<?>** invece di un tipo grezzo. Se passate un tipo grezzo a un metodo che usa **<?>** è possibile che il compilatore deduca l'effettivo parametro di tipo, in modo che il metodo possa aggirare l'ostacolo e chiamare un altro metodo che utilizza il tipo esatto. Il codice che segue dimostra questa tecnica, chiamata "conversione dell'intercattazione" (*capture conversion*) perché il tipo di metacarattere non specificato viene "intercettato" e convertito in un tipo esatto. In questo esempio, i commenti sugli avvertimenti si applicano soltanto se l'annotazione **@SuppressWarnings** viene rimossa:

```
//: generics/CaptureConversion.java

public class CaptureConversion {
  static <T> void f1(Holder<T> holder) {
    T t = holder.get();
    System.out.println(t.getClass().getSimpleName());
  }
  static void f2(Holder<?> holder) {
    f1(holder); // Chiamata con tipo intercettato
  }
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    Holder raw = new Holder<Integer>(1);
    // f1(raw); // Visualizza avvertimenti
    f2(raw); // Nessun avvertimento
    Holder rawBasic = new Holder();
    rawBasic.set(new Object()); // Avvertimento
    f2(rawBasic); // Nessun avvertimento
    // Upcast a Holder<?>, riesce a capirlo:
```

```
    Holder<?> wildcarded = new Holder<Double>(1.0);
    f2(wildcarded);
  }
} /* Output:
Integer
Object
Double
*///:~
```

I parametri di tipo in **f1()** sono tutti esatti, senza metacaratteri né limiti; in **f2()** il parametro **Holder** è un metacarattere senza limiti, che sembrerebbe quindi effettivamente sconosciuto. Tuttavia, in **f2()** viene chiamato **f1()** e quest'ultimo richiede un parametro noto. Ciò che accade è che il tipo di parametro viene intercettato nel corso della chiamata a **f2()**, in modo da essere utilizzabile nella chiamata a **f1()**.

Potreste domandarvi se questa tecnica potrebbe essere utilizzata anche per la scrittura: la risposta è no, perché richiederebbe il passaggio di un tipo specifico con **Holder<?>**. La conversione dell'intercettazione funziona soltanto nelle situazioni in cui all'interno del metodo occorra lavorare con il tipo esatto. Tenete presente che non è possibile che **f2()** restituisca **T**, poiché **T** è ignoto a **f2()**. La conversione dell'intercettazione è una tecnica interessante, ma piuttosto limitata.

> **Esercizio 29** (5) Create un metodo generico che accetti come argomento **Holder<List<?>>**. Determinate i metodi richiamabili e non richiamabili per **Holder** e per **List** e ripetete l'esercizio con un argomento **List<Holder<?>>**.

## Problemi

Questo paragrafo esamina un insieme di problemi eterogenei che possono verificarsi nell'utilizzo dei generici Java.

### *Nessun primitivo come parametro di tipo*

Come già accennato in questo capitolo, una delle limitazioni dei generici Java è l'impossibilità di utilizzare i tipi primitivi come parametri di tipo: per esempio, non potete creare un **ArrayList<int>**.

La soluzione consiste nell'utilizzare i tipi wrapper e la funzionalità autoboxing di Java SE5. Se create un **ArrayList<Integer>** e utilizzate primitivi **int** con

questo contenitore, scoprirete che autoboxing esegue automaticamente la conversione da e verso **Integer**, il che è quasi come avere un **ArrayList<int>**:

```
//: generics/ListOfInt.java
// L'autoboxing compensa l'impossibilita' di usare
// i tipi primitivi nei generici.
import java.util.*;

public class ListOfInt {
  public static void main(String[] args) {
    List<Integer> li = new ArrayList<Integer>();
    for(int i = 0; i < 5; i++)
      li.add(i);
    for(int i : li)
      System.out.print(i + " ");
  }
} /* Output:
0 1 2 3 4
*///:~
```

Ricordate che l'autoboxing permette anche la sintassi foreach per produrre **int**.

In generale questa soluzione funziona bene, consentendovi di registrare e richiamare valori **int**. Sembrano esservi implicate alcune conversioni, ma queste non sono visibili. Tuttavia, nel caso in cui le prestazioni rappresentino un problema, potrete utilizzare una variante specializzata dei contenitori adattata per lavorare con i tipi primitivi; una versione open source è **org.apache.commons.collections.primitives**.

Di seguito è mostrato un altro meccanismo, che genera un **Set** di **Byte**:

```
//: generics/ByteSet.java
import java.util.*;

public class ByteSet {
  Byte[] possibles = { 1,2,3,4,5,6,7,8,9 };
  Set<Byte> mySet =
    new HashSet<Byte>(Arrays.asList(possibles));
```

```
    // Ma non e' permesso usare:
    // Set<Byte> mySet2 = new HashSet<Byte>(
    //   Arrays.<Byte>asList(1,2,3,4,5,6,7,8,9));
  } ///:~
```

Tenete presente che l'autoboxing risolve alcuni problemi, ma non tutti. Il seguente esempio mostra un'interfaccia generica **Generator** che specifica un metodo **next()**, il quale restituisce un oggetto del tipo di parametro. La classe **FArray** contiene un metodo generico che si serve di un generatore per popolare un array di oggetti: in questo caso non è possibile rendere generica la classe, poiché il metodo è **static**. Le implementazioni di **Generator** saranno trattate nel Capitolo 4, e in **main()** potete vedere **FArray.fill()** utilizzato per popolare array di oggetti:

```
//: generics/PrimitiveGenericTest.java
import net.mindview.util.*;

// Popola un array utilizzando un generatore:
class FArray {
  public static <T> T[] fill(T[] a, Generator<T> gen) {
    for(int i = 0; i < a.length; i++)
      a[i] = gen.next();
    return a;
  }
}

public class PrimitiveGenericTest {
  public static void main(String[] args) {
    String[] strings = FArray.fill(
      new String[7], new RandomGenerator.String(10));
    for(String s : strings)
      System.out.println(s);
    Integer[] integers = FArray.fill(
      new Integer[7], new RandomGenerator.Integer());
    for(int i: integers)
      System.out.println(i);
    // L'autoboxing non puo' aiutarvi in questo caso.
    // Questo non compila:
```

```
    // int[] b =
    //   FArray.fill(new int[7], new RandIntGenerator());
  }
} /* Output:
YNzbrnyGcF
OWZnTcQrGs
eGZMmJMRoE
suEcUOneOE
dLsmwHLGEa
hKcxrEqUCB
bkInaMesbt
7052
6665
2654
3909
5202
2209
5458
*///:~
```

Poiché **RandomGenerator.Integer** implementa **Generator<Integer>**, l'autore era convinto che l'autoboxing convertisse automaticamente il valore di **next()** da **Integer** a **int**. Purtroppo l'autoboxing non si applica agli array, quindi questo meccanismo non funziona.

> **Esercizio 30** (2) Create un **Holder** per ogni wrapper di tipi primitivi e mostrate che l'autoboxing e l'autounboxing funziona con i metodi **set()** e **get()** di ciascun **Holder**.

## *Implementare interfacce parametrizzate*

Una classe non può implementare due varianti della stessa interfaccia generica, perché a causa della cancellazione queste diventano entrambe la stessa interfaccia. Ecco una situazione in cui si verifica questa "collisione":

```
//: generics/MultipleInterfaceVariants.java
// {CompileTimeError} (Non compilera')

interface Payable<T> {}
```

```
class Employee implements Payable<Employee> {}
class Hourly extends Employee
  implements Payable<Hourly> {} ///:~
```

**Hourly** non compilerà, perché la cancellazione riduce **Payable<Employee>** e **Payable<Hourly>** alla stessa classe, **Payable**, quindi il codice di questo esempio si tradurrebbe in una duplice implementazione della stessa interfaccia. Tuttavia, è interessante notare che rimuovendo i parametri generici da entrambi gli utilizzi di **Payable**, come fa il compilatore durante la cancellazione, il codice compila.

Come vedrete nel prosieguo del capitolo, questo problema può diventare spiacevole se dovete gestire alcune delle interfacce fondamentali di Java, quali **Comparable**.

> **Esercizio 31** (1) Rimuovete tutti i generici da **MultipleInterfaceVariants.java** e modificate il codice in modo che l'esempio compili.

## *Casting e avvertimenti del compilatore*

L'utilizzo di un cast o di **instanceof** con un parametro di tipo generico non ha alcun effetto. Il seguente contenitore registra internamente i valori come **Object** e li converte nuovamente in **T** al momento del loro prelievo:

```
//: generics/GenericCast.java

class FixedSizeStack<T> {
  private int index = 0;
  private Object[] storage;
  public FixedSizeStack(int size) {
    storage = new Object[size];
  }
  public void push(T item) { storage[index++] = item; }
  @SuppressWarnings("unchecked")
  public T pop() { return (T)storage[--index]; }
}

public class GenericCast {
  public static final int SIZE = 10;
  public static void main(String[] args) {
```

```
    FixedSizeStack<String> strings =
      new FixedSizeStack<String>(SIZE);
    for(String s : "A B C D E F G H I J".split(" "))
      strings.push(s);
    for(int i = 0; i < SIZE; i++) {
      String s = strings.pop();
      System.out.print(s + " ");
    }
  }
} /* Output:
J I H G F E D C B A
*///:~
```

Senza l'annotazione **@SuppressWarnings**, il compilatore produrrà un avvertimento "unchecked cast" per il metodo **pop()**. A causa della cancellazione non potete sapere se il cast è sicuro, ma il metodo **pop()** in realtà non esegue alcun casting. La **T** viene cancellata al suo primo limite, che in modalità predefinita è **Object**, pertanto in pratica **pop()** si limita a convertire un **Object** in un **Object**.

Vi sono situazioni in cui i generici non eliminano la necessità di eseguire un cast: questo genera un avviso del compilatore, che è una condizione inadeguata. Per esempio:

```
//: generics/NeedCasting.java
import java.io.*;
import java.util.*;

public class NeedCasting {
  @SuppressWarnings("unchecked")
  public void f(String[] args) throws Exception {
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream(args[0]));
    List<Widget> shapes = (List<Widget>)in.readObject();
  }
} ///:~
```

Come vedrete nel capitolo successivo, **readObject()** non può sapere che cosa sta leggendo, per cui restituisce un oggetto che deve essere convertito. Se però

commentate l'annotazione **@SuppressWarnings** e compilate il programma, il compilatore visualizzerà un avvertimento:

```
Note: NeedCasting.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Se seguite le istruzioni e ricompilate con l'opzione **-Xlint:unchecked**, otterrete:

```
NeedCasting.java:10: warning: [unchecked] unchecked cast
found   : java.lang.Object
required: java.util.List<Widget>
    List<Shape> shapes = (List<Widget>)in.readObject();
```

Siete quindi costretti a eseguire un cast, ma nello stesso tempo siete informati che non dovreste eseguirlo. Per risolvere questo dilemma dovete utilizzare una nuova forma di cast introdotta con Java SE5, il cast tramite classe generica:

```
//: generics/ClassCasting.java
import java.io.*;
import java.util.*;

public class ClassCasting {
  @SuppressWarnings("unchecked")
  public void f(String[] args) throws Exception {
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream(args[0]));
      // Non compilera':
//    List<Widget> lw1 =
//    List<Widget>.class.cast(in.readObject());
    List<Widget> lw2 = List.class.cast(in.readObject());
  }
} ///:~
```

Non potete tuttavia convertire al tipo effettivo, **List<Widget>**. In pratica non potete scrivere:

```
List<Widget>.class.cast(in.readObject())
```

e anche se aggiungete un altro cast come questo

```
(List<Widget>)List.class.cast(in.readObject())
```

otterrete di nuovo un avvertimento.

**Esercizio 32** (1) Verificate che **FixedSizeStack** in **GenericCast.java** genera eccezioni se tentate di uscire dai suoi limiti. Questo indica forse che il controllo dei limiti non è richiesto?

**Esercizio 33** (3) Fate funzionare **GenericCast.java** servendovi di un **ArrayList**.

## *Sovraccarico*

Questo codice non compilerà, sebbene sia ragionevole pensare che lo faccia:

```
//: generics/UseList.java
// {CompileTimeError} (Non compilera')
import java.util.*;

public class UseList<W,T> {
  void f(List<T> v) {}
  void f(List<W> v) {}
} ///:~
```

A causa della cancellazione, sovraccaricando il metodo si ottiene la stessa segnatura.

Dovrete invece fornire nomi del metodo distinti, quando gli argomenti cancellati non producono un elenco di argomenti univoco:

```
//: generics/UseList2.java
import java.util.*;

public class UseList2<W,T> {
  void f1(List<T> v) {}
  void f2(List<W> v) {}
} ///:~
```

Fortunatamente, questo genere di problema viene rilevato dal compilatore.

## *Classe di base che dirotta un'interfaccia*

Supponete di avere una classe **Pet** che sia "comparabile" (**Comparable**) con altri oggetti **Pet**:

```
//: generics/ComparablePet.java

public class ComparablePet
implements Comparable<ComparablePet> {
  public int compareTo(ComparablePet arg) { return 0; }
} ///:~
```

Potrebbe avere senso tentare di limitare il tipo con cui può essere confrontata una sottoclasse di **ComparablePet**. Per esempio, un **Cat** dovrebbe essere **Comparable** soltanto con altri **Cat**:

```
//: generics/HijackedInterface.java
// {CompileTimeError} (Non compilera')

class Cat extends ComparablePet implements Comparable<Cat>{
  // Errore: Comparable cannot be inherited with
  // different arguments: <Cat> and <Pet>
  public int compareTo(Cat arg) { return 0; }
} ///:~
```

Sfortunatamente questa tecnica non funziona. Quando per **Comparable** è impostato l'argomento **ComparablePet**, nessun'altra classe implementata può essere più confrontata con qualcosa di diverso da **ComparablePet**:

```
//: generics/RestrictedComparablePets.java

class Hamster extends ComparablePet
implements Comparable<ComparablePet> {
  public int compareTo(ComparablePet arg) { return 0; }
}

// O semplicemente:

class Gecko extends ComparablePet {
  public int compareTo(ComparablePet arg) { return 0; }
} ///:~
```

L'esempio con la classe **Hamster** mostra che è possibile reimplementare la stessa interfaccia che è disponibile in **ComparablePet**, purché sia *esattamente* la stessa, inclusi i tipi di parametro.

Questo meccanismo è analogo alla sovrascrittura dei metodi nella classe di base, come si può vedere per la classe **Gecko**.

## Tipi autolimitati

Esiste una forma idiomatica complessa che appare regolarmente nei generici Java. Eccone un esempio:

```
class SelfBounded<T extends SelfBounded<T>> { // ...
```

Questa tecnica ha l'effetto vertiginoso di due specchi rivolti uno verso l'altro, una sorta di riflessione infinita: la classe **SelfBounded** accetta un argomento generico **T**, **T** è limitato da un vincolo e quest'ultimo è **SelfBounded**, con **T** come argomento.

È difficile analizzare questa sintassi la prima volta che la si incontra, un'ulteriore prova del fatto che la parola chiave **extends**, utilizzata con i limiti, ha un comportamento assolutamente diverso rispetto a quando viene utilizzata per generare le sottoclassi.

### *Generici stranamente ricorsivi*

Per comprendere il significato di un tipo autolimitato è opportuno iniziare con una versione più semplice dell'idioma, priva di autolimitazione.

Non vi è possibile ereditare direttamente da un parametro generico, tuttavia *potete* ereditare da una classe che si serve di questo parametro generico nella propria definizione. Potete quindi scrivere:

```
//: generics/CuriouslyRecurringGeneric.java

class GenericType<T> {}

public class CuriouslyRecurringGeneric
  extends GenericType<CuriouslyRecurringGeneric> {} ///:~
```

Questa sintassi potrebbe essere chiamata *generici stranamente ricorsivi* (*Curiously Recurring Generics*, CRG), sulla falsariga della presentazione del pat-

tern *Curiously Recurring Template* di Jim Coplien, scritto in C++. L'aspetto "curiosamente ricorsivo" si riferisce al fatto che la classe compare, in modo decisamente strano, nella propria classe di base.

Per capire che cosa significa, provate a tradurre questo concetto in linguaggio umano: "creo una nuova classe che eredita da un tipo generico che accetta il nome della mia classe come suo parametro". Che cosa può fare il tipo di base generico una volta assegnato il nome alla classe derivata? Considerato che i generici lavorano su argomenti e tipi di ritorno, Java può generare una classe di base che utilizza il tipo derivato come propri argomenti e tipi di ritorno, e persino servirsi del tipo derivato per i tipi di campo, anche se questi saranno cancellati e trasformati in **Object**. Ecco un esempio di classe generica che esprime questo concetto:

```
//: generics/BasicHolder.java

public class BasicHolder<T> {
  T element;
  void set(T arg) { element = arg; }
  T get() { return element; }
  void f() {
    System.out.println(element.getClass().getSimpleName());
  }
} ///:~
```

Questo è un tipo generico normale con metodi che accettano e producono oggetti del tipo di parametro specificato, con un metodo che opera sul campo archiviato (**element**), benché su questo campo esegua solo operazioni **Object**.

Potete utilizzare **BasicHolder** in un generico stranamente ricorsivo (CRG):

```
//: generics/CRGWithBasicHolder.java

class Subtype extends BasicHolder<Subtype> {}

public class CRGWithBasicHolder {
  public static void main(String[] args) {
    Subtype st1 = new Subtype(), st2 = new Subtype();
    st1.set(st2);
```

```
    Subtype st3 = st1.get();
    st1.f();
  }
} /* Output:
Subtype
*///:~
```

In questo esempio potete notare qualcosa di importante: la nuova classe **Subtype** accetta argomenti e restituisce valori di tipo **Subtype**, non semplicemente della classe di base **BasicHolder**. Questa è l'essenza del CRG: *la classe di base sostituisce la classe derivata per i suoi parametri*. Questo significa che la classe di base generica si trasforma in una specie di modello per le funzionalità comuni a tutte le sue classi derivate, ma questa funzionalità utilizzerà il tipo derivato per tutti i suoi argomenti e valori di ritorno: nella classe risultante, in pratica, verrà utilizzato il tipo esatto anziché il tipo di base. Quindi, in **Subtype** sia l'argomento di **set()** sia il tipo di ritorno di **get()** sono esattamente **Subtype**.

## *Autolimitazione*

Come vedete nell'esempio seguente, **BasicHolder** può servirsi di qualunque tipo come proprio parametro generico:

```
//: generics/Unconstrained.java

class Other {}
class BasicOther extends BasicHolder<Other> {}

public class Unconstrained {
  public static void main(String[] args) {
    BasicOther b = new BasicOther(), b2 = new BasicOther();
    b.set(new Other());
    Other other = b.get();
    b.f();
  }
} /* Output:
Other
*///:~
```

L'autolimitazione (*selfbounding*) implementa il passo supplementare di *forzare* il generico a essere utilizzato come proprio argomento di limite. Osservate come la classe risultante può essere utilizzata e come non può esserlo:

```
//: generics/SelfBounding.java

class SelfBounded<T extends SelfBounded<T>> {
  T element;
  SelfBounded<T> set(T arg) {
    element = arg;
    return this;
  }
  T get() { return element; }
}

class A extends SelfBounded<A> {}
class B extends SelfBounded<A> {} // Anche questo e' OK

class C extends SelfBounded<C> {
  C setAndGet(C arg) { set(arg); return get(); }
}

class D {}
// Non potete fare questo:
// class E extends SelfBounded<D> {}
// Errore di compilazione:
// Type parameter D is not within its bound

// Potete fare questo, percio' non potete forzare la forma
// idiomatica:
class F extends SelfBounded {}

public class SelfBounding {
  public static void main(String[] args) {
    A a = new A();
    a.set(new A());
    a = a.set(new A()).get();
```

```
    a = a.get();
    C c = new C();
    c = c.setAndGet(new C());
  }
} ///:~
```

L'autolimitazione richiede l'utilizzo della classe in un rapporto di ereditarietà come il seguente:

```
class A extends SelfBounded<A> {}
```

Questo vi costringe a passare la classe in fase di definizione come parametro della classe di base.

Vi chiederete quale sia il valore aggiunto ottenuto dall'autolimitazione del parametro. La risposta è semplice: fare in modo che il parametro di tipo sia lo stesso della classe in corso di definizione. Come risulta chiaro nella definizione della classe **B**, potete anche ereditare da una classe **SelfBounded** che utilizza un parametro di un'altra **SelfBounded**, sebbene l'utilizzo predominante sembri essere quello illustrato per la classe **A**. Il tentativo di definire **E** mostra che non è permesso servirsi di un tipo di parametro che non sia **SelfBounded**.

Purtroppo, come vedete, **F** compila senza avvertimenti, e questo significa che la forma idiomatica autolimitante non può essere forzata. Se ciò fosse realmente importante, potreste ricorrere a un'utility esterna per accertarvi che i tipi grezzi non vengano utilizzati al posto dei tipi parametrizzati.

Tenete presente che potete rimuovere il vincolo e tutte le classi compileranno, ma compilerà anche la classe **E**:

```
//: generics/NotSelfBounded.java

public class NotSelfBounded<T> {
  T element;
  NotSelfBounded<T> set(T arg) {
    element = arg;
    return this;
  }
  T get() { return element; }
```

```
}

class A2 extends NotSelfBounded<A2> {}
class B2 extends NotSelfBounded<A2> {}

class C2 extends NotSelfBounded<C2> {
  C2 setAndGet(C2 arg) { set(arg); return get(); }
}

class D2 {}
// Ora e' OK:
class E2 extends NotSelfBounded<D2> {} ///:~
```

Chiaramente, quindi, il vincolo autolimitante serve soltanto per forzare il rapporto di ereditarietà. Servendovi di questa tecnica, avrete la certezza che il parametro di tipo utilizzato dalla classe sarà lo stesso tipo di base della classe che sta utilizzando quel parametro, forzando chiunque si serva di quella classe a seguire questa forma.

È anche possibile applicare l'autolimitazione ai metodi generici:

```
//: generics/SelfBoundingMethods.java

public class SelfBoundingMethods {
  static <T extends SelfBounded<T>> T f(T arg) {
    return arg.set(arg).get();
  }
  public static void main(String[] args) {
    A a = f(new A());
  }
} ///:~
```

Questo codice fa sì che il metodo si applichi soltanto a un argomento autolimitato nella forma indicata.

## Covarianza degli argomenti

Il valore dei tipi autolimitanti è che producono *tipi di argomento covarianti*, ovvero si adattano in funzione delle sottoclassi.

Dal momento che in Java SE5 sono stati introdotti i *tipi di ritorno covarianti*, non è così importante che i tipi autolimitanti producano anche tipi di ritorno identici a quelli della sottoclasse: potete ottenere questo risultato servendovi di tipi di ritorno covarianti.

```
//: generics/CovariantReturnTypes.java

class Base {}
class Derived extends Base {}

interface OrdinaryGetter {
  Base get();
}

interface DerivedGetter extends OrdinaryGetter {
  // E' permesso che il tipo di ritorno del metodo
  // sovrascritto vari:
  Derived get();
}

public class CovariantReturnTypes {
  void test(DerivedGetter d) {
    Derived d2 = d.get();
  }
} ///:~
```

Il metodo **get()** in **DerivedGetter** sovrascrive il metodo **get()** di **OrdinaryGetter** *e* restituisce un tipo derivato dal tipo restituito da **OrdinaryGetter.get()**. Pur essendo perfettamente logico che un metodo di tipo derivato sia in grado di restituire un tipo più specifico del tipo (classe) di base che sovrascrive, nelle precedenti versioni di Java questa operazione non era ammessa.

In effetti un generico autolimitato produce il tipo derivato esatto come un valore di ritorno, come potete vedere con **get()**:

```
//: generics/GenericsAndReturnTypes.java

interface GenericGetter<T extends GenericGetter<T>> {
  T get();
```

```
}

interface Getter extends GenericGetter<Getter> {}

public class GenericsAndReturnTypes {
  void test(Getter g) {
    Getter result = g.get();
    GenericGetter gg = g.get(); // Anche la classe di base
  }
} ///:~
```

Notate che questo codice compila perché i tipi di ritorno covarianti sono stati inclusi in Java SE5.

Nel codice non generico, tuttavia, i tipi di argomento non possono essere realizzati per variare con i sottotipi:

```
//: generics/OrdinaryArguments.java

class OrdinarySetter {
  void set(Base base) {
    System.out.println("OrdinarySetter.set(Base)");
  }
}

class DerivedSetter extends OrdinarySetter {
  void set(Derived derived) {
    System.out.println("DerivedSetter.set(Derived)");
  }
}

public class OrdinaryArguments {
  public static void main(String[] args) {
    Base base = new Base();
    Derived derived = new Derived();
    DerivedSetter ds = new DerivedSetter();
    ds.set(derived);
    ds.set(base); // Compila: sovraccaricato,
```

```
                    // non sovrascritto!
  }
} /* Output:
DerivedSetter.set(Derived)
OrdinarySetter.set(Base)
*///:~
```

Le istruzioni **set(derived)** e **set(base)** sono entrambe ammesse, per cui **DerivedSetter.set()** non sta sovrascrivendo **OrdinarySetter.set()**, ma lo sta *sovraccaricando*. Dall'output potete vedere che in **DerivedSetter** vi sono due metodi: la versione della classe di base è ancora disponibile, e questo dimostra che è stata sovraccaricata.

Con i tipi autolimitanti nella classe derivata, tuttavia, c'è un solo metodo che considera come suo argomento il tipo derivato e non quello di base:

```
//: generics/SelfBoundingAndCovariantArguments.java

interface SelfBoundSetter<T extends SelfBoundSetter<T>> {
  void set(T arg);
}

interface Setter extends SelfBoundSetter<Setter> {}

public class SelfBoundingAndCovariantArguments {
  void testA(Setter s1, Setter s2, SelfBoundSetter sbs) {
    s1.set(s2);
    // s1.set(sbs); // Errore:
    // set(Setter) in SelfBoundSetter<Setter>
    // cannot be applied to (SelfBoundSetter)
  }
} ///:~
```

Il compilatore non riconosce il tentativo di passare il tipo di base come argomento di **set()**, poiché non c'è un metodo con quella segnatura: in realtà l'argomento è stato sovrascritto.

Senza autolimitazione interviene il meccanismo di ereditarietà standard che attiva l'overloading, proprio come nel caso non generico:

```
//: generics/PlainGenericInheritance.java

class GenericSetter<T> { // Non auto-limitante
  void set(T arg){
    System.out.println("GenericSetter.set(Base)");
  }
}

class DerivedGS extends GenericSetter<Base> {
  void set(Derived derived){
    System.out.println("DerivedGS.set(Derived)");
  }
}

public class PlainGenericInheritance {
  public static void main(String[] args) {
    Base base = new Base();
    Derived derived = new Derived();
    DerivedGS dgs = new DerivedGS();
    dgs.set(derived);
    dgs.set(base); // Compila: sovraccaricato,
                   // non sovrascritto!
  }
} /* Output:
DerivedGS.set(Derived)
GenericSetter.set(Base)
*///:~
```

Questo codice imita **OrdinaryArguments.java**: in quell'esempio **DerivedSetter** eredita da **OrdinarySetter**, che contiene un **set(Base)**. In questo caso, **DerivedGS** eredita da **GenericSetter** che contiene anch'esso un **set(Base)**, creato dal generico. Esattamente come in **OrdinaryArguments.java**, l'output mostra che **DerivedGS** contiene due versioni sovraccariche di **set()**. Senza autolimitazione avreste sovraccaricato i tipi di argomento; grazie all'autolimitazione, invece, vi ritrovate con una sola versione del metodo, che accetta il tipo di argomento esatto.

**Esercizio 34** (4) Create un tipo generico autolimitato contenente un metodo astratto, che accetta come argomento un parametro di tipo generico e produce un valore di ritorno del parametro di tipo generico. In un metodo non astratto della classe chiamate il metodo astratto e restituite il relativo risultato, poi ereditate dal tipo autolimitato e testate la classe risultante.

## Sicurezza di tipo dinamica

Poiché è possibile passare contenitori generici al codice precedente Java SE5, esiste ancora la possibilità che vecchio codice possa alterare i vostri contenitori. In Java SE5, la libreria **java.util.Collections** offre un insieme di programmi di utilità per risolvere il problema del controllo di tipo in queste situazioni: i metodi **static checkedCollection()**, **checkedList()**, **checkedMap()**, **checkedSet()**, **checkedSortedMap()** e **checkedSortedSet()**. Ciascuno di essi accetta come primo argomento il contenitore da controllare dinamicamente e, come secondo argomento, il tipo da far rispettare.

Un contenitore controllato solleva una **ClassCastException** nel momento in cui provate a *inserire* un oggetto improprio, diversamente da un contenitore pre-generico (grezzo) che vi informa del problema quando *estraete* l'oggetto. In quest'ultimo caso sapete che c'è un problema ma non sapete quale ne sia la causa, mentre con i contenitori controllati potete scoprire che cosa ha cercato di inserire l'oggetto sbagliato.

Ora esaminerete il problema di "inserire un gatto in un elenco di cani" ricorrendo a un contenitore controllato. In questo esempio **oldStyleMethod()** rappresenta il vecchio codice perché prende una **List** grezza, mentre l'annotazione **@SuppressWarnings("unchecked")** è necessaria per sopprimere l'avvertimento che ne risulterà:

```
//: generics/CheckedList.java
// Utilizzo di Collection.checkedList().
import typeinfo.pets.*;
import java.util.*;

public class CheckedList {
  @SuppressWarnings("unchecked")
  static void oldStyleMethod(List probablyDogs) {
    probablyDogs.add(new Cat());
```

```
  }
  public static void main(String[] args) {
    List<Dog> dogs1 = new ArrayList<Dog>();
    oldStyleMethod(dogs1); // Accetta tranquillamente un Cat
    List<Dog> dogs2 = Collections.checkedList(
      new ArrayList<Dog>(), Dog.class);
    try {
      oldStyleMethod(dogs2); // Solleva un'eccezione
    } catch(Exception e) {
      System.out.println(e);
    }
    // I tipi derivati funzionano bene:
    List<Pet> pets = Collections.checkedList(
      new ArrayList<Pet>(), Pet.class);
    pets.add(new Dog());
    pets.add(new Cat());
  }
} /* Output:
java.lang.ClassCastException: Attempt to insert class
typeinfo.pets.Cat element into collection with element type
class typeinfo.pets.Dog
*///:~
```

Quando eseguite il programma, vedete che l'inserimento di un **Cat** funziona tranquillamente con la **List dogs1**, mentre **dogs2** solleva immediatamente un'eccezione, segnalando l'inserimento di un tipo errato. Potete anche vedere che è possibile inserire oggetti di tipi derivati in un contenitore controllato che sta controllando la classe di base.

**Esercizio 35** (1) Modificate **CheckedList.java** in modo che utilizzi le classi **Coffee** definite in questo capitolo.

## Eccezioni

A causa della cancellazione, l'utilizzo dei generici con le eccezioni è estremamente limitato. Una clausola **catch** non può intercettare un'eccezione di un tipo generico, poiché il tipo esatto di quest'ultima deve essere noto sia in fase di compilazione sia durante l'esecuzione; inoltre una classe generica non

può ereditare (né direttamente né indirettamente) da **Throwable**, e questo costituisce ulteriore impedimento nel provare a definire eccezioni generiche che non potrebbero comunque essere intercettate.

È tuttavia possibile utilizzare i parametri di tipo nella clausola **throws** di una dichiarazione di metodo. Questo consente di scrivere codice generico che varia in funzione del tipo di eccezione controllata:

```
//: generics/ThrowGenericException.java
import java.util.*;

interface Processor<T,E extends Exception> {
  void process(List<T> resultCollector) throws E;
}

class ProcessRunner<T,E extends Exception>
extends ArrayList<Processor<T,E>> {
  List<T> processAll() throws E {
    List<T> resultCollector = new ArrayList<T>();
    for(Processor<T,E> processor : this)
      processor.process(resultCollector);
    return resultCollector;
  }
}

class Failure1 extends Exception {}

class Processor1 implements Processor<String,Failure1> {
  static int count = 3;
  public void
  process(List<String> resultCollector) throws Failure1 {
    if(count-- > 1)
      resultCollector.add("Hep!");
    else
      resultCollector.add("Ho!");
    if(count < 0)
       throw new Failure1();
  }
```

```
}

class Failure2 extends Exception {}

class Processor2 implements Processor<Integer,Failure2> {
  static int count = 2;
  public void
  process(List<Integer> resultCollector) throws Failure2 {
    if(count-- == 0)
      resultCollector.add(47);
    else {
      resultCollector.add(11);
    }
    if(count < 0)
       throw new Failure2();
  }
}

public class ThrowGenericException {
  public static void main(String[] args) {
    ProcessRunner<String,Failure1> runner =
      new ProcessRunner<String,Failure1>();
    for(int i = 0; i < 3; i++)
      runner.add(new Processor1());
    try {
      System.out.println(runner.processAll());
    } catch(Failure1 e) {
      System.out.println(e);
    }

    ProcessRunner<Integer,Failure2> runner2 =
      new ProcessRunner<Integer,Failure2>();
    for(int i = 0; i < 3; i++)
      runner2.add(new Processor2());
    try {
      System.out.println(runner2.processAll());
```

```
      } catch(Failure2 e) {
        System.out.println(e);
      }
    }
  } ///:~
```

Un oggetto **Processor** esegue un metodo **process()** e potrebbe sollevare un'eccezione di tipo **E**. Il risultato di **process()** è registrato in **resultCollector List<T>**: questo è chiamato *parametro di raccolta* (*collecting parameter*). Un oggetto **ProcessRunner** possiede un metodo **processAll()**, il quale esegue ogni oggetto **Process** che contiene e restituisce il **resultCollector**.

Se non potete parametrizzare le eccezioni che vengono sollevate, non sarete in grado di scrivere in modo generico questo codice per via delle eccezioni controllate.

> **Esercizio 36** (2) Aggiungete una seconda eccezione parametrizzata alla classe **Processor** e dimostrate che le eccezioni possono variare in modo indipendente.

## Mixin

Con il passare del tempo il termine *mixin* o *mix-in* sembra acquistare sempre più significati, ma il concetto di base si riferisce alla "miscelazione" delle capacità di diverse classi per produrre una classe-risultato che rappresenti tutti i tipi inclusi nel mixin. Questa spesso è un'operazione che eseguirete all'ultimo minuto, il che la rende pratica per un rapido "assemblaggio" delle classi.

Uno dei vantaggi dei mixin è che applicano in modo coerente le caratteristiche e i comportamenti a tutte le classi implicate. Come ulteriore beneficio, se modificate qualcosa in una delle classi del mixin il cambiamento si irradierà a tutte le classi che compongono il mixin. In considerazione di tale comportamento, i mixin assumono connotazioni tipiche della cosiddetta *programmazione orientata agli aspetti* (*AOP*, A*spect-Oriented Programming*) e gli aspetti sono spesso proposti per risolvere il problema del mixin.

### *Mixin in C++*

Una delle argomentazioni più forti a favore dell'ereditarietà multipla in C++ è l'utilizzo dei mixin. In ogni caso una tecnica più interessante ed elegante ai mixin prevede l'adozione dei tipi parametrizzati, secondo i quali un mixin è una classe che eredita dal relativo parametro di tipo. In C++ potete generare

facilmente mixin, perché il linguaggio si ricorda il tipo dei relativi parametri di modello (*template parameter*).

Considerate questo esempio di codice C++ con due tipi di mixin: uno che permette di mescolare la proprietà di avere un timestamp e un altro che mescola un numero di serie per ogni istanza dell'oggetto:

```
//: generics/Mixins.cpp
#include <string>
#include <ctime>
#include <iostream>
using namespace std;

template<class T> class TimeStamped : public T {
  long timeStamp;
public:
  TimeStamped() { timeStamp = time(0); }
  long getStamp() { return timeStamp; }
};

template<class T> class SerialNumbered : public T {
  long serialNumber;
  static long counter;
public:
  SerialNumbered() { serialNumber = counter++; }
  long getSerialNumber() { return serialNumber; }
};

// Definisce e inizializza l'archiviazione statica:
template<class T> long SerialNumbered<T>::counter = 1;

class Basic {
  string value;
public:
  void set(string val) { value = val; }
  string get() { return value; }
};
```

```
int main() {
  TimeStamped<SerialNumbered<Basic> > mixin1, mixin2;
  mixin1.set("test string 1");
  mixin2.set("test string 2");
  cout << mixin1.get() << " " << mixin1.getStamp() <<
    " " << mixin1.getSerialNumber() << endl;
  cout << mixin2.get() << " " << mixin2.getStamp() <<
    " " << mixin2.getSerialNumber() << endl;
} /* Output: (Esempio)
test string 1 1129840250 1
test string 2 1129840250 2
*///:~
```

In **main()**, il tipo risultante di **mixin1** e **mixin2** possiede tutti i metodi dei tipi mescolati. Potete considerare un mixin come una funzione che fa corrispondere le classi esistenti alle nuove sottoclassi. Notate quanto sia semplice generare un mixin servendovi di questa tecnica; in pratica vi limitate a ordinare quello che vi occorre, che appare come per incanto:

```
TimeStamped<SerialNumbered<Basic> > mixin1, mixin2;
```

Purtroppo i generici di Java non offrono una funzionalità così avanzata. La cancellazione dimentica il tipo della classe di base, di conseguenza una classe generica non può ereditare direttamente da un parametro generico.

## *Mixin con le interfacce*

Una soluzione spesso raccomandata è il ricorso alle interfacce per produrre l'effetto dei mixin, come in questo esempio:

```
//: generics/Mixins.java
import java.util.*;

interface TimeStamped { long getStamp(); }

class TimeStampedImp implements TimeStamped {
  private final long timeStamp;
  public TimeStampedImp() {
```

```
    timeStamp = new Date().getTime();
  }
  public long getStamp() { return timeStamp; }
}

interface SerialNumbered { long getSerialNumber(); }

class SerialNumberedImp implements SerialNumbered {
  private static long counter = 1;
  private final long serialNumber = counter++;
  public long getSerialNumber() { return serialNumber; }
}

interface Basic {
  public void set(String val);
  public String get();
}

class BasicImp implements Basic {
  private String value;
  public void set(String val) { value = val; }
  public String get() { return value; }
}

class Mixin extends BasicImp
implements TimeStamped, SerialNumbered {
  private TimeStamped timeStamp = new TimeStampedImp();
  private SerialNumbered serialNumber =
    new SerialNumberedImp();
  public long getStamp() { return timeStamp.getStamp(); }
  public long getSerialNumber() {
    return serialNumber.getSerialNumber();
  }
}

public class Mixins {
```

```
    public static void main(String[] args) {
      Mixin mixin1 = new Mixin(), mixin2 = new Mixin();
      mixin1.set("test string 1");
      mixin2.set("test string 2");
      System.out.println(mixin1.get() + " " +
        mixin1.getStamp() +  " " + mixin1.getSerialNumber());
      System.out.println(mixin2.get() + " " +
        mixin2.getStamp() +  " " + mixin2.getSerialNumber());
    }
  } /* Output: (Esempio)
  test string 1 1132437151359 1
  test string 2 1132437151359 2
  *///:~
```

In pratica la classe **Mixin** sta utilizzando la funzionalità di *delega*, pertanto ogni tipo componente il mixin deve avere un campo in **Mixin**, e la classe dovrà essere dotata di tutti i metodi necessari per trasmettere le chiamate all'oggetto opportuno. Questo esempio si serve di classi banali, ma tenete presente che con un mixin più complesso il codice aumenta rapidamente di dimensioni.[2]

**Esercizio 37** (2) Aggiungete una nuova classe mixin **Colored** a **Mixins.java**, mescolatela con **Mixin** e dimostrate che funziona.

## *Utilizzo del modello Decorator*

Se analizzate il modo in cui viene implementato, il concetto di mixin sembra strettamente connesso al design pattern **Decorator**.[3]

I decorator (letteralmente "decoratori") sono utilizzati di frequente quando, per soddisfare ogni combinazione possibile, la semplice *estensione* (*subclassing* o *eredità di implementazione*) produce così tante classi da diventare poco pratica.

Il pattern *Decorator* si serve di oggetti "stratificati", integrando i diversi oggetti con nuove responsabilità in modo dinamico e trasparente. Il decoratore

2. Alcuni ambienti di programmazione, quali Eclipse e IDEA di IntelliJ, producono automaticamente il codice di delega.
3. I modelli sono l'argomento del volume *Thinking in Patterns (with Java)*, che potete procurarvi all'indirizzo *www.mindview.net*. Un altro testo di riferimento è *Design Patterns*, di Erich Gamma *et al.* (Addison-Wesley, 1995).

specifica che tutti gli oggetti i quali inglobano il vostro oggetto iniziale hanno la stessa interfaccia di base. Un oggetto può essere sottoposto a "decorazione", e diventa così *decoratable*: gli si integrano nuove funzionalità, incorporandolo in altre classi. Questo rende trasparente l'utilizzo dei decoratori: esiste un insieme di messaggi comuni che potete trasmettere a un oggetto, che sia stato decorato o meno. Un classe di decorazione ("decorante") può anche aggiungere metodi, ma come vedrete, questa possibilità è limitata.

I decoratori vengono implementati ricorrendo alla composizione e alle strutture formali (la gerarchia *decoratable/decorator*, letteralmente "decorabile-decoratore"), mentre i mixin sono basati sull'ereditarietà. Potete quindi considerare i mixin, basati sui tipi parametrizzati, come una sorta di funzionalità di decorazione che non richiede la struttura di ereditarietà prevista dal design pattern Decorator. L'esempio precedente può essere rimaneggiato servendosi di Decorator:

```
//: generics/decorator/Decoration.java
package generics.decorator;
import java.util.*;

class Basic {
private String value;
  public void set(String val) { value = val; }
  public String get() { return value; }
}

class Decorator extends Basic {
  protected Basic basic;
  public Decorator(Basic basic) { this.basic = basic; }
  public void set(String val) { basic.set(val); }
  public String get() { return basic.get(); }
}

class TimeStamped extends Decorator {
  private final long timeStamp;
  public TimeStamped(Basic basic) {
    super(basic);
    timeStamp = new Date().getTime();
  }
```

```
  public long getStamp() { return timeStamp; }
}

class SerialNumbered extends Decorator {
  private static long counter = 1;
  private final long serialNumber = counter++;
  public SerialNumbered(Basic basic) { super(basic); }
  public long getSerialNumber() { return serialNumber; }
}

public class Decoration {
  public static void main(String[] args) {
    TimeStamped t = new TimeStamped(new Basic());
    TimeStamped t2 = new TimeStamped(
      new SerialNumbered(new Basic()));
    //! t2.getSerialNumber(); // Non disponibile
    SerialNumbered s = new SerialNumbered(new Basic());
    SerialNumbered s2 = new SerialNumbered(
      new TimeStamped(new Basic()));
    //! s2.getStamp(); // Non disponibile
  }
} ///:~
```

La classe che deriva da un mixin contiene tutti i metodi cui siete interessati, ma il tipo dell'oggetto che risulta dall'utilizzo dei decoratori è l'ultimo tipo con cui è stato decorato. In altre parole, benché sia possibile aggiungere più livelli, il tipo effettivo è dato dal livello finale, quindi soltanto i metodi del livello finale sono visibili; il tipo del mixin racchiude invece tutti i tipi che sono stati mescolati. Un inconveniente notevole del Decorator, pertanto, è che funziona efficacemente solo con uno strato di decorazione (quello finale), mentre la tecnica di mixin è nettamente più naturale. In definitiva, l'utilizzo dei Decorator è una soluzione limitata al problema dei mixin.

**Esercizio 38** (4) Create un semplice sistema Decorator iniziando da un **Coffee** di base, quindi fornite decoratori per il latte caldo, la schiuma di latte, il cioccolato, il caramello e la panna montata.

## Mixin con proxy dinamici

È possibile utilizzare un proxy dinamico per produrre un meccanismo che modelli i mixin più strettamente di quanto faccia il Decorator: consultate il capitolo precedente per maggiori informazioni sul funzionamento sui proxy dinamici. Con un proxy dinamico, il tipo dinamico della classe risultante è rappresentato dai tipi combinati che sono stati mescolati.

Tenuto conto dei vincoli dei proxy dinamici, ogni classe che viene aggiunta deve essere l'implementazione di un'interfaccia:

```
//: generics/DynamicProxyMixin.java
import java.lang.reflect.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Tuple.*;

class MixinProxy implements InvocationHandler {
  Map<String,Object> delegatesByMethod;
  public MixinProxy(TwoTuple<Object,Class<?>>... pairs) {
    delegatesByMethod = new HashMap<String,Object>();
    for(TwoTuple<Object,Class<?>> pair : pairs) {
      for(Method method : pair.second.getMethods()) {
        String methodName = method.getName();
        // La prima interfaccia nella mappa implementa
        // il metodo.
        if (!delegatesByMethod.containsKey(methodName))
          delegatesByMethod.put(methodName, pair.first);
      }
    }
  }
  public Object invoke(Object proxy, Method method,
    Object[] args) throws Throwable {
    String methodName = method.getName();
    Object delegate = delegatesByMethod.get(methodName);
    return method.invoke(delegate, args);
  }
  @SuppressWarnings("unchecked")
```

```
  public static Object newInstance(TwoTuple... pairs) {
    Class[] interfaces = new Class[pairs.length];
    for(int i = 0; i < pairs.length; i++) {
      interfaces[i] = (Class)pairs[i].second;
    }
    ClassLoader cl =
      pairs[0].first.getClass().getClassLoader();
    return Proxy.newProxyInstance(
      cl, interfaces, new MixinProxy(pairs));
  }
}

public class DynamicProxyMixin {
  public static void main(String[] args) {
    Object mixin = MixinProxy.newInstance(
      tuple(new BasicImp(), Basic.class),
      tuple(new TimeStampedImp(), TimeStamped.class),
      tuple(new SerialNumberedImp(),SerialNumbered.class));
    Basic b = (Basic)mixin;
    TimeStamped t = (TimeStamped)mixin;
    SerialNumbered s = (SerialNumbered)mixin;
    b.set("Hello");
    System.out.println(b.get());
    System.out.println(t.getStamp());
    System.out.println(s.getSerialNumber());
  }
} /* Output: (Esempio)
Hello
1132519137015
1
*///:~
```

Poiché *soltanto* il tipo dinamico include tutti i tipi coinvolti, questa funzionalità non è ancora così pratica come la tecnica adottata in C++, perché vi costringe a eseguire il downcast al tipo appropriato prima di poter chiamarne i metodi. In ogni caso, è significativamente più vicina a un mixin reale.

L'adattamento dei mixin al linguaggio Java ha richiesto una quantità ingente di lavoro, inclusa la creazione di un add-on, il linguaggio Jam, specifico per il supporto ai mixin.

**Esercizio 39** (1) Aggiungete una nuova classe mixin **Colored** a **DynamicProxyMixin.java**, includetela nel **mixin** e dimostrate che funziona.

## Latent typing

All'inizio di questo capitolo, vi è stata presentata l'idea di scrivere codice che possa essere applicato il più genericamente possibile. Per fare questo vi occorrono tecniche per allentare i vincoli sui tipi con i quali funziona il vostro codice, senza perdere i vantaggi del controllo di tipo statico. In questo modo avrete la possibilità di scrivere codice utilizzabile in diverse situazioni senza modifiche, ovvero più "generico".

I generici Java sembrano rappresentare un ulteriore passo in questa direzione. Quando scrivete o utilizzate generici che si limitano a contenere oggetti, il codice funziona con qualunque tipo tranne i primitivi, anche se, come avete visto, l'autoboxing attenua questo problema. Da un diverso punto di vista, è come se i "contenitori" generici potessero affermare: "non importa di quale tipo siete". Il codice che non si cura del tipo con cui lavora può essere infatti applicato ovunque, ed è quindi veramente "generico".

Avete anche visto che, quando occorre eseguire manipolazioni sui tipi generici diverse dalla chiamata di metodi **Object**, si presenta un problema: la cancellazione richiede infatti che specifichiate i limiti dei tipi generici utilizzabili, per chiamare in modo sicuro i metodi specifici per gli oggetti generici presenti nel codice. Questa è una limitazione notevole al concetto di "generico", poiché implica la necessità di sottoporre i tipi generici a vincoli, in modo che ereditino da particolari classi o implementino particolari interfacce. In alcuni casi potreste ritrovarvi a utilizzare una classe o un'interfaccia standard, perché un generico limitato potrebbe non essere diverso da una normale classe o interfaccia.

Una soluzione offerta da alcuni linguaggi di programmazione è chiamata *latent typing* o *structural typing*, oppure, in modo più stravagante, *duck typing,* un'espressione piuttosto diffusa, forse perché non sottintende il bagaglio storico che caratterizza altra terminologia.

Di norma, il codice generico chiama soltanto alcuni metodi su un tipo generico; un linguaggio con funzionalità di latent typing, invece, allenta il vincolo (e produce codice più generico) richiedendo soltanto l'implementazione di un sottoinsieme di metodi, *non* di una classe o di un'interfaccia particolari.

Per questo motivo il *latent typing* vi permette di "attraversare" le gerarchie delle classi, chiamando metodi che non fanno parte di un'interfaccia comune. Così, in effetti, una porzione di codice potrebbe dire: "Non importa di quale tipo siete, purché possiate parlare (**speak()**) e sedervi (**sit()**)". Non dovendo contare su un tipo specifico, il vostro codice sarà più generico.

Il latent typing è un meccanismo di organizzazione e riutilizzo del codice, e vi permette di scrivere una porzione di codice che può essere riutilizzata più facilmente. L'organizzazione e il riutilizzo del codice sono le leve fondamentali di tutta la programmazione: scrivilo una volta, utilizzalo più volte e mantieni il codice in un punto. Non essendo costretti a dare un nome a un'interfaccia esatta su cui opera il codice, con il *latent typing* potete scrivere una quantità inferiore di codice e applicarlo facilmente in più punti.

Due esempi di linguaggi che supportano il latent typing sono Python, che potete scaricare gratuitamente da *www.python.org*, e C++.[4]

Python è un linguaggio a scrittura dinamica, in cui praticamente tutti i controlli sui tipi avvengono al momento dell'esecuzione; C++, al contrario, è un linguaggio a scrittura statica nel quale il controllo sui tipi avviene in fase di compilazione. Il latent typing, invece, non richiede controllo di tipo statico o dinamico.

Se prendete la descrizione citata in precedenza e la esprimete in Python, otterrete:

```
#: generics/DogsAndRobots.py

class Dog:
    def speak(self):
        print "Arf!"
    def sit(self):
        print "Sitting"
    def reproduce(self):
        pass

class Robot:
    def speak(self):
        print "Clic!"
    def sit(self):
```

4. Anche i linguaggi Ruby e Smalltalk supportano il latent typing.

```
        print "Clac!"
    def oilChange(self):
        pass
def perform(anything):
    anything.speak()
    anything.sit()

a = Dog()
b = Robot()
perform(a)
perform(b)
#:~
```

Il linguaggio Python si serve della cosiddetta indentazione (rientri) per determinare l'ambito, quindi non richiede parentesi graffe; i due punti (**:**) segnalano l'inizio di un nuovo ambito. Il simbolo '**#**' indica un commento fino all'estremità della riga, come // in Java. I metodi di una classe specificano esplicitamente l'equivalente del riferimento **this** come primo argomento, chiamato **self** per convenzione. Le chiamate ai costruttori non richiedono alcuna parola chiave di tipo "**new**". Python, inoltre, consente l'utilizzo delle funzioni normali (non membro), come dimostra l'utilizzo di **perform()**.

In **perform(anything)** avrete notato che **anything** non ha un tipo, ma rappresenta un semplice identificativo: deve potere eseguire le operazioni richieste da **perform()** e pertanto implica un'interfaccia, che tuttavia non deve essere scritta esplicitamente perché è *latente*. Il metodo **perform()** non si cura del tipo del suo argomento, e questo consente di passare al metodo qualunque oggetto supporti i metodi **speak()** e **sit()**. Se passate a **perform()** un oggetto che non ammette queste operazioni, otterrete un'eccezione in fase di esecuzione.

Potete produrre lo stesso effetto in C++:

```
//: generics/DogsAndRobots.cpp

class Dog {
public:
  void speak() {}
  void sit() {}
  void reproduce() {}
```

```
};

class Robot {
public:
  void speak() {}
  void sit() {}
  void oilChange() {
};

template<class T> void perform(T anything) {
  anything.speak();
  anything.sit();
}

int main() {
  Dog d;
  Robot r;
  perform(d);
  perform(r);
} ///:~
```

Sia in Python sia in C++ **Dog** e **Robot** non hanno niente in comune, tranne il fatto che possiedono due metodi con segnature identiche: dal punto di vista dei tipi, si tratta di tipi completamente diversi. Tuttavia **perform()** non si cura del tipo specifico del suo argomento, e il latent typing gli consente di accettare entrambi i tipi di oggetti.

Invece C++ si accerta di poter realmente trasmettere questi messaggi, e se provate a passare un tipo errato il compilatore produrrà un messaggio di errore: tenete presente che storicamente questi messaggi di errore sono prolissi e minacciosi, e sono la ragione principale per cui i template di C++ hanno una così scarsa reputazione. Anche se eseguono queste operazioni in tempi diversi, C++ in sede di compilazione e Python al momento dell'esecuzione, entrambi i linguaggi si accertano che i tipi siano utilizzati correttamente e per questo motivo sono considerati linguaggi *strongly typed* (con controllo dei tipi forte).[5]

5. Poiché permette di utilizzare i cast, che in pratica disabilitano il sistema di controllo dei tipi, alcuni sostengono che C++ sia un linguaggio *weakly typed* (con controllo dei tipi debole): l'autore ritiene tuttavia che tale affermazione sia eccessiva.

## Il latent typing non compromette lo strong typing

Poiché i generici sono stati aggiunti a Java soltanto in fasi successive, non è stato possibile implementare la funzionalità di latent typing, che di conseguenza Java non supporta. Per questo motivo, a una prima occhiata sembra che il meccanismo dei generici Java sia "meno generico" rispetto a un linguaggio che supporta il latent typing.[6]

Per esempio, se provate a implementare l'esempio precedente in Java sarete costretti a utilizzare una classe o un'interfaccia, e a specificarle in un'espressione con limiti:

```
//: generics/Performs.java

public interface Performs {
  void speak();
  void sit();
} ///:~

//: generics/DogsAndRobots.java
// Java non supporta il latent typing
import typeinfo.pets.*;
import static net.mindview.util.Print.*;

class PerformingDog extends Dog implements Performs {
  public void speak() { print("Woof!"); }
  public void sit() { print("Sitting"); }
  public void reproduce() {}
}

class Robot implements Performs {
  public void speak() { print("Clic!"); }
  public void sit() { print("Clac!"); }
  public void oilChange() {}
}
```

6. L'implementazione dei generici Java che si serve della cancellazione viene spesso descritta con l'espressione "tipi generici di *seconda classe*".

```
class Communicate {
  public static <T extends Performs>
  void perform(T performer) {
    performer.speak();
    performer.sit();
  }
}

public class DogsAndRobots {
  public static void main(String[] args) {
    PerformingDog d = new PerformingDog();
    Robot r = new Robot();
    Communicate.perform(d);
    Communicate.perform(r);
  }
} /* Output:
Woof!
Sitting
Clic!
Clac!
*///:~
```

Tenete presente, però, che il funzionamento del metodo **perform()** non richiede necessariamente l'utilizzo dei generici, ma è sufficiente specificare che dovrà accettare un oggetto **Performs**:

```
//: generics/SimpleDogsAndRobots.java
// Rimozione del generico; il codice continua a funzionare.

class CommunicateSimply {
  static void perform(Performs performer) {
    performer.speak();
    performer.sit();
  }
}

public class SimpleDogsAndRobots {
```

```
    public static void main(String[] args) {
      CommunicateSimply.perform(new PerformingDog());
      CommunicateSimply.perform(new Robot());
    }
  } /* Output:
  Woof!
  Sitting
  Clic!
  Clac!
  *///:~
```

In questo caso i generici non erano necessari, dal momento che le classi sono già costrette a implementare l'interfaccia **Performs**.

## Compensazione alla mancanza di latent typing

Il fatto che Java sia sprovvisto di latent typing non significa che il vostro codice generico limitato non possa applicarsi a diverse gerarchie di tipo: in effetti è ancora possibile generare codice veramente generico, sebbene questo richieda un certo impegno supplementare.

### *Riflessione*

Una tecnica che potete utilizzare è la *riflessione*. Ecco un metodo **perform()** che utilizza il latent typing:

```
//: generics/LatentReflection.java
// Utilizzo della riflessione per riprodurre la funzionalità
// di latent typing.
import java.lang.reflect.*;
import static net.mindview.util.Print.*;

// Non implementa Performs:
class Mime {
  public void walkAgainstTheWind() {}
  public void sit() { print("Pretending to sit"); }
  public void pushInvisibleWalls() {}
  public String toString() { return "Mime"; }
```

```
}

// Non implementa Performs:
class SmartDog {
  public void speak() { print("Woof!"); }
  public void sit() { print("Sitting"); }
  public void reproduce() {}
}

class CommunicateReflectively {
  public static void perform(Object speaker) {
    Class<?> spkr = speaker.getClass();
    try {
      try {
        Method speak = spkr.getMethod("speak");
        speak.invoke(speaker);
      } catch(NoSuchMethodException e) {
        print(speaker + "cannot speak");
      }
      try {
        Method sit = spkr.getMethod("sit");
        sit.invoke(speaker);
      } catch(NoSuchMethodException e) {
        print(speaker + "cannot sit");
      }
    } catch(Exception e) {
      throw new RuntimeException(speaker.toString(), e);
    }
  }
}

public class LatentReflection {
  public static void main(String[] args) {
    CommunicateReflectively.perform(new SmartDog());
    CommunicateReflectively.perform(new Robot());
    CommunicateReflectively.perform(new Mime());
```

```
  }
} /* Output:
Woof!
Sitting
Clic!
Clac!
Mime cannot speak
Pretending to sit
*///:~
```

In questo codice le classi sono del tutto scollegate e non hanno classi di base (tranne **Object**), né interfacce in comune. Grazie alla riflessione, **CommunicateReflectively.perform()** è in grado di stabilire in modo dinamico se i metodi richiesti sono disponibili e può chiamarli; può persino gestire il fatto che **Mime** ha soltanto uno dei metodi necessari, e soddisfa parzialmente il suo compito.

## *Applicare un metodo a una sequenza*

La riflessione fornisce alcune possibilità interessanti, ma relega tutto il controllo dei tipi a runtime, e per questo motivo non è desiderabile in molte situazioni: se poteste implementare questo controllo al momento della compilazione sarebbe decisamente meglio. Ma è possibile fare coesistere il controllo dei tipi in fase di compilazione e il *latent typing*?

Considerate un esempio che evidenzia il problema: supponete di volere creare un metodo **apply()** che applichi qualsiasi metodo a tutti gli oggetti presenti in una sequenza; questa è una situazione in cui le interfacce non sembrano essere adatte. Volete applicare qualsiasi metodo a una collezione di oggetti e le interfacce vi limitano eccessivamente nel descrivere "qualsiasi metodo". Come risolverete questo problema in Java?

Inizialmente potreste affrontare il problema ricorrendo alla tecnica di riflessione, che risulta essere piuttosto elegante grazie agli argomenti variabili (varарg) di Java SE5:

```
//: generics/Apply.java
// {main: ApplyTest}
import java.lang.reflect.*;
import java.util.*;
import static net.mindview.util.Print.*;
```

```
public class Apply {
  public static <T, S extends Iterable<? extends T>>
  void apply(S seq, Method f, Object... args) {
    try {
      for(T t: seq)
        f.invoke(t, args);
    } catch(Exception e) {
      // I fallimenti sono dovuti a errori del programmatore
      throw new RuntimeException(e);
    }
  }
}

class Shape {
  public void rotate() { print(this + " ruota"); }
  public void resize(int newSize) {
    print(this + " resize " + newSize);
  }
}

class Square extends Shape {}

class FilledList<T> extends ArrayList<T> {
  public FilledList(Class<? extends T> type, int size) {
    try {
      for(int i = 0; i < size; i++)
        // Considera il costruttore predefinito:
        add(type.newInstance());
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
}

class ApplyTest {
  public static void main(String[] args) throws Exception {
    List<Shape> shapes = new ArrayList<Shape>();
```

```
    for(int i = 0; i < 10; i++)
      shapes.add(new Shape());
    Apply.apply(shapes, Shape.class.getMethod("rotate"));
    Apply.apply(shapes,
      Shape.class.getMethod("resize", int.class), 5);
    List<Square> squares = new ArrayList<Square>();
    for(int i = 0; i < 10; i++)
      squares.add(new Square());
    Apply.apply(squares, Shape.class.getMethod("rotate"));
    Apply.apply(squares,
      Shape.class.getMethod("resize", int.class), 5);

    Apply.apply(new FilledList<Shape>(Shape.class, 10),
      Shape.class.getMethod("rotate"));
Apply.apply(new FilledList<Shape>(Square.class, 10),
      Shape.class.getMethod("rotate"));

    SimpleQueue<Shape> shapeQ = new SimpleQueue<Shape>();
    for(int i = 0; i < 5; i++) {
      shapeQ.add(new Shape());
      shapeQ.add(new Square());
    }
    Apply.apply(shapeQ, Shape.class.getMethod("rotate"));
  }
} /* (Da eseguire per visualizzare l'output) *///:~
```

Per eseguire questo programma, occorre digitare **java ApplyTest.**

In **Apply.java** la fortuna vi assiste, perché Java offre un'interfaccia **Iterable** utilizzata dalla libreria di contenitori. Per questo motivo il metodo **apply()** può accettare qualsiasi cosa implementi l'interfaccia **Iterable**, che include tutte le classi di **Collection** come **List**; può tuttavia anche accettare qualsiasi altro oggetto venga reso **Iterable**, per esempio la classe **SimpleQueue** definita in questo contesto e utilizzata in precedenza in **main()**:

```
//: generics/SimpleQueue.java
// Un tipo diverso di container Iterable
import java.util.*;
```

```
public class SimpleQueue<T> implements Iterable<T> {
  private LinkedList<T> storage = new LinkedList<T>();
  public void add(T t) { storage.offer(t); }
  public T get() { return storage.poll(); }
  public Iterator<T> iterator() {
    return storage.iterator();
  }
} ///:~
```

In **Apply.java** le eccezioni sono convertite in **RuntimeExceptions** perché non avrebbe senso ripristinarle, considerato che in questo caso rappresentano errori commessi dal programmatore.

Notate che è stato necessario applicare limiti e metacaratteri al fine di potere utilizzare **Apply** e **FilledList** nelle situazioni opportune: provate a eliminare limiti e metacaratteri, e vedrete che alcune applicazioni di **Apply** e **FilledList** non funzioneranno.

Il vero dilemma è rappresentato da **FilledList**. Affinché sia possibile utilizzare un tipo deve esserci un costruttore predefinito, cioè privo di argomenti: Java non ha modo di verificare questa condizione al momento della compilazione, che pertanto si trasforma in un problema a runtime. Una tecnica comune per garantire il controllo in sede di compilazione è definire un'interfaccia factory, con un metodo che genera oggetti; a quel punto **FilledList** accetterebbe quell'interfaccia anziché la "factory grezza" del token di tipo. Il problema di questa soluzione è che tutte le classi utilizzate in **FilledList** dovranno implementare la vostra interfaccia factory. Purtroppo la maggior parte delle classi che viene generata non è al corrente della vostra interfaccia, e quindi non la implementa. In seguito vedrete una soluzione che si serve degli adattatori.

Tuttavia il metodo indicato, che ricorre a un token di tipo, probabilmente è un compromesso ragionevole, quantomeno come soluzione temporanea: mediante questo approccio l'utilizzo di una classe **FilledList** è abbastanza facile da poter essere applicato, piuttosto che ignorato. Naturalmente, dal momento che gli errori vengono segnalati in fase di esecuzione, dovete poter contare sul fatto che questi errori appaiano il più presto possibile nel processo di sviluppo.

Tenete presente che la tecnica del token di tipo è raccomandata nella letteratura Java, in particolare nel documento sui generici di Gilad Bracha, citato alla conclusione del capitolo, in cui si fa osservare che: "È una forma idiomatica utilizzata in modo diffuso, per esempio nelle nuove API per la manipolazione delle annotazioni".

Tuttavia l'autore ha rilevato una certa discontinuità di apprezzamento da parte dei programmatori nei confronti di questa tecnica: alcuni preferiscono adottare l'approccio factory, descritto in precedenza in questo capitolo.

Inoltre, per quanto elegante possa essere la soluzione Java, occorre rilevare che l'utilizzo della riflessione, sebbene notevolmente migliorato nelle recenti versioni di Java, può rivelarsi più lento di un'implementazione che non tiene conto della riflessione, dal momento che richiede una notevole elaborazione in fase di esecuzione. Queste considerazioni non dovrebbero impedirvi di ricorrere a questa tecnica, almeno come prima soluzione e per non diventare preda di un'ottimizzazione prematura; è innegabile tuttavia che esista una distinzione tra i due metodi.

> **Esercizio 40** (3) Aggiungete un metodo **speak()** a tutti gli animali in **typeinfo.pets**, poi modificate **Apply.java** per chiamare il metodo **speak()** per una collezione eterogenea di **Pet**.

## *Come fare quando non si dispone dell'interfaccia corretta*

L'esempio precedente ha potuto trarre beneficio da un'interfaccia **Iterable** già integrata, che era esattamente ciò di cui avevate bisogno. Ma come fare in un caso più generale, quando non esiste un'interfaccia adatta alle vostre esigenze?

Per esempio, provate a generalizzare il concetto in **FilledList** allo scopo di generare un metodo parametrizzato **fill()** che accetti una sequenza e la popoli per mezzo di un **Generator**. Se provate a tradurre questo concetto in codice Java avrete un problema, perché non esiste un'interfaccia "**Addable**" adatta e analoga all'interfaccia **Iterable** dell'esempio precedente. Quindi, anziché indicare "qualsiasi cosa per cui potete chiamare **add()**", dovrete dire "un sottotipo di **Collection**". Il codice risultante non sarà particolarmente generico, perché deve funzionare soltanto con le implementazioni di **Collection**. Se cercate di utilizzare una classe che non implementa **Collection**, il codice generico non funzionerà. Esso potrebbe essere simile all'esempio seguente:

```
//: generics/Fill.java
// Generalizzazione del concetto di FilledList
// {main: FillTest}
import java.util.*;

// Non funziona con "nulla che abbia un add()." Non essendoci
// un'interfaccia "aggiungibile" ("Addable") siete costretti a
```

```
// utilizzare una Collection. In questo caso non e' possibile
// generalizzare utilizzando i generici.

public class Fill {
  public static <T> void fill(Collection<T> collection,
  Class<? extends T> classToken, int size) {
    for(int i = 0; i < size; i++)
      // Considera il costruttore predefinito:
      try {
        collection.add(classToken.newInstance());
      } catch(Exception e) {
        throw new RuntimeException(e);
      }
  }
}

class Contract {
  private static long counter = 0;
  private final long id = counter++;
  public String toString() {
    return getClass().getName() + " " + id;
  }
}
class TitleTransfer extends Contract {}

class FillTest {
  public static void main(String[] args) {
    List<Contract> contracts = new ArrayList<Contract>();
    Fill.fill(contracts, Contract.class, 3);
    Fill.fill(contracts, TitleTransfer.class, 2);
    for(Contract c: contracts)
      System.out.println(c);
    SimpleQueue<Contract> contractQueue =
      new SimpleQueue<Contract>();
    // Non funzionera', perche' fill() non e' abbastanza
    // generico:
```

```
    // Fill.fill(contractQueue, Contract.class, 3);
  }
} /* Output:
Contract 0
Contract 1
Contract 2
TitleTransfer 3
TitleTransfer 4
*///:~
```

Per eseguire questo programma, occorre digitare **java FillTest**.

È in occasioni come questa che si rivela utile il meccanismo di un tipo parametrizzato con il latent typing, perché non vi lascia alla mercé di decisioni progettuali precedenti di un qualsiasi programmatore di libreria: non vi costringe quindi a riscrivere il codice ogni volta che incontrate una nuova libreria che non tiene conto della vostra situazione, permettendovi di realizzare codice veramente "generico". Nel caso in questione, considerato che (comprensibilmente) i progettisti Java non hanno avvertito l'esigenza di creare un'interfaccia "**Addable**", siete limitati nell'ambito della gerarchia **Collection** e **SimpleQueue**, che pure possiede un metodo **add()**, non funzionerà. Essendo costretto a lavorare con **Collection** il codice non sarebbe particolarmente "generico"; con il latent typing, al contrario, non sarebbe così.

## Simulare il latent typing mediante gli adattatori

Ora sapete che i generici di Java non dispongono della funzionalità di latent typing, e avrete perciò bisogno di qualcosa di analogo per scrivere codice che vada oltre i limiti di classe, vale a dire codice "generico". Esiste un modo per ovviare a questa limitazione?

Come si comporterebbe il latent typing in questo caso? Significherebbe poter scrivere codice che dica: "Non mi interessa il tipo utilizzato in questo contesto, purché abbia questi metodi". In effetti il latent typing genera un'*interfaccia implicita* contenente i metodi richiesti. Ne consegue che scrivendo voi stessi l'interfaccia necessaria, tenuto conto che Java non lo fa automaticamente, dovreste riuscire a risolvere il problema.

Il codice necessario per produrre l'interfaccia che desiderate da una che avete a disposizione è un esempio di utilizzo del design pattern Adapter. Potete servirvi degli adattatori per adeguare le classi esistenti in modo che produca-

no l'interfaccia voluta, con una quantità di codice relativamente minima. La soluzione, che utilizza la gerarchia **Coffee** definita in precedenza, dimostra i diversi modi per realizzare adattatori:

```
//: generics/Fill2.java
// Utilizzo degli adattatori per simulare il latent typing.
// {main: Fill2Test}
import generics.coffee.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

interface Addable<T> { void add(T t); }

public class Fill2 {
  // Versione classtoken:
  public static <T> void fill(Addable<T> addable,
  Class<? extends T> classToken, int size) {
    for(int i = 0; i < size; i++)
      try {
        addable.add(classToken.newInstance());
      } catch(Exception e) {
        throw new RuntimeException(e);
      }
  }
  // Versione con generatore:
  public static <T> void fill(Addable<T> addable,
  Generator<T> generator, int size) {
    for(int i = 0; i < size; i++)
      addable.add(generator.next());
  }
}

// Per adattare un tipo di base, occorre usare
// la composizione.
// Rendete Addable qualsiasi Collection utilizzando
// la composizione:
```

```
class AddableCollectionAdapter<T> implements Addable<T> {
  private Collection<T> c;
  public AddableCollectionAdapter(Collection<T> c) {
    this.c = c;
  }
  public void add(T item) { c.add(item); }
}

// Un Helper per intercettare automaticamente il tipo:
class Adapter {
  public static <T>
  Addable<T> collectionAdapter(Collection<T> c) {
    return new AddableCollectionAdapter<T>(c);
  }
}

// Per adattare un tipo specifico, potete usare
// l'ereditarieta'.
// Rendete Addable una SimpleQueue utilizzando
// l'ereditarieta':
class AddableSimpleQueue<T>
extends SimpleQueue<T> implements Addable<T> {
  public void add(T item) { super.add(item); }
}

class Fill2Test {
  public static void main(String[] args) {
    // Adatta una Collection:
    List<Coffee> carrier = new ArrayList<Coffee>();
    Fill2.fill(
      new AddableCollectionAdapter<Coffee>(carrier),
      Coffee.class, 3);
    // Metodo Helper per intercettare il tipo:
    Fill2.fill(Adapter.collectionAdapter(carrier),
      Latte.class, 2);
    for(Coffee c: carrier)
```

```
      print(c);
    print("----------------------");
    // Utilizza una classe adattata:
    AddableSimpleQueue<Coffee> coffeeQueue =
      new AddableSimpleQueue<Coffee>();
    Fill2.fill(coffeeQueue, Mocha.class, 4);
    Fill2.fill(coffeeQueue, Latte.class, 1);
    for(Coffee c: coffeeQueue)
      print(c);
  }
} /* Output:
Coffee 0
Coffee 1
Coffee 2
Latte 3
Latte 4
----------------------
Mocha 5
Mocha 6
Mocha 7
Mocha 8
Latte 9
*///:~
```

Per eseguire questo programma, occorre digitare **java Fill2Test**.

A differenza di quello di **Fill** il codice di **Fill2** non richiede una **Collection**, ma soltanto qualcosa che implementi **Addable**, e **Addable** è stato scritto soltanto per **Fill**: questa è una manifestazione del tipo latente che sarebbe stato auspicabile il compilatore avesse realizzato da sé.

In questa versione è stato anche aggiunto un metodo **fill()** sovraccaricato che accetta un **Generator**, anziché un token di tipo. Il **Generator** garantisce la sicurezza dei tipi al momento della compilazione: il compilatore si accerta che gli venga passato un **Generator** adeguato, pertanto non può essere sollevata alcuna eccezione.

Il primo adattatore, **AddableCollectionAdapter**, funziona con il tipo di base **Collection**, il che significa che potrà essere utilizzata qualsiasi implementa-

zione di **Collection**. Questa versione si limita a registrare il riferimento a **Collection**, che utilizza per implementare **add()**.

Se anziché la classe di base di una gerarchia disponete di un tipo specifico, potrete risparmiare una parte del codice di creazione dell'adattatore ricorrendo all'ereditarietà, come vedete in **AddableSimpleQueue**.

Il metodo **Fill2Test.main()** mostra i vari tipi di adattatori al lavoro. In primo luogo, un tipo **Collection** viene adattato mediante **AddableCollectionAdapter**. Una seconda versione di questa **Collection** si serve di un metodo di supporto generico, e potete notare come quest'ultimo catturi il tipo in modo che non debba essere scritto esplicitamente: un trucco pratico per produrre codice più elegante.

In seguito viene utilizzata la classe **AddableSimpleQueue** pre-adattata. Osservate che in entrambi i casi gli adattatori permettono l'utilizzo di **Fill2.fill()** con classi che in precedenza non implementavano **Addable**.

L'utilizzo di adattatori come questo sembrerebbe compensare la mancanza di latent typing, e permettere così di scrivere codice veramente generico. Tuttavia rappresenta pur sempre un livello aggiuntivo che dovrà essere compreso sia dal creatore della libreria sia dal programmatore utente, un concetto che non tutti sono in grado di afferrare prontamente, in particolare i programmatori con minore esperienza. Rimuovendo questo livello supplementare il latent typing semplifica l'applicazione del codice generico, ed è in questo che risiede il suo valore.

**Esercizio 41** (1) Modificate **Fill2.java** affinché utilizzi le classi in **typeinfo.pets** in luogo delle classi **Coffee**.

## Strategia degli oggetti funzione

Questo esempio finale creerà vero codice generico utilizzando il meccanismo dell'adattatore descritto nel paragrafo precedente. L'esempio è iniziato come tentativo di generare una somma da una sequenza di elementi di qualsiasi tipo che sia addizionabile, ma si è evoluto al punto da eseguire operazioni generiche implementando uno stile di programmazione *funzionale*.

Se analizzate unicamente il processo per provare ad aggiungere gli oggetti, noterete che questo è il tipico caso in cui si hanno operazioni comuni a più classi, ma tali operazioni non sono rappresentate in alcuna classe di base che possa essere specificata: talvolta è possibile utilizzare persino un operatore '**+**', altre volte potrebbe essere disponibile una sorta di metodo 'add'. Questa è la classica situazione che incontrate quando provate a scrivere il codice generico, perché vorreste che il codice fosse applicabile a più classi (in

particolare, come in questo caso, classi multiple che già esistono e non sono "adattabili"). Anche se fosse possibile limitare questo caso alle sottoclassi di **Number**, tale superclasse non avrebbe nessuna capacità di essere "addable".

La soluzione consiste nel design pattern *Strategy*, che produce codice più elegante perché isola completamente le "cose che cambiano" all'interno di un oggetto funzione.[7]

Un oggetto funzione si comporta in un certo senso come una funzione (metodo): in generale, ogni oggetto funzione ha soltanto un metodo utile. Tenete presente che i linguaggi i quali supportano il sovraccarico degli operatori permettono di chiamare questo metodo come un qualsiasi metodo standard. Il beneficio degli oggetti funzione rispetto ai metodi ordinari è che possono essere passati come argomenti ad altri oggetti e continuare a mantenere uno stato persistente tra le varie chiamate. Ovviamente è possibile ottenere un comportamento analogo con il metodo di una classe ma, come per qualsiasi pattern, l'oggetto funzione si distingue soprattutto per la sua intenzionalità. In questo caso l'intento è quello di creare un oggetto il quale si comporti come un singolo metodo che sia possibile passare come argomento ad altri oggetti; in questo senso il modello è strettamente connesso con il pattern Strategy, da cui spesso è persino indistinguibile.

Come avrete modo di constatare con diversi modelli, a questo punto il codice diventa confuso: state generando oggetti funzione che eseguono un adattamento e che vengono passati a metodi per essere utilizzati come strategie.

Adottando questo approccio, aggiungete i vari tipi di metodi generici che in origine avevate pensato di generare e altri. Ecco il risultato:

```
//: generics/Functional.java
import java.math.*;
import java.util.concurrent.atomic.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Tipi diversi di oggetti funzione:
interface Combiner<T> { T combine(T x, T y); }
interface UnaryFunction<R,T> { R function(T x); }
```

7. Talvolta gli oggetti funzione sono chiamati anche *funtori* (*functor*). L'autore ha scelto di utilizzare l'espressione "oggetto funzione", in quanto il termine *funtore* in matematica ha un significato diverso.

```
interface Collector<T> extends UnaryFunction<T,T> {
  T result(); // Estrae il risultato del parametro di raccolta
}
interface UnaryPredicate<T> { boolean test(T x); }

public class Functional {
  // Chiama l'oggetto Combiner su ogni elemento per combinarlo
  // con un risultato progressivo, che alla fine viene
  // restituito:
  public static <T> T
  reduce(Iterable<T> seq, Combiner<T> combiner) {
    Iterator<T> it = seq.iterator();
    if(it.hasNext()) {
      T result = it.next();
      while(it.hasNext())
        result = combiner.combine(result, it.next());
      return result;
    }
    // Se seq e' la lista vuota:
    return null; // Oppure solleva un'eccezione
  }
  // Accetta un oggetto funzione e lo chiama per ogni oggetto
  // in elenco, ignorando il valore di ritorno. L'oggetto
  // funzione può agire come parametro di raccolta, pertanto
  // alla fine viene restituito.
  public static <T> Collector<T>
  forEach(Iterable<T> seq, Collector<T> func) {
    for(T t : seq)
      func.function(t);
    return func;
  }
  // Crea una lista di risultati chiamando un oggetto funzione
  // per ogni oggetto nella List:
  public static <R,T> List<R>
  transform(Iterable<T> seq, UnaryFunction<R,T> func) {
    List<R> result = new ArrayList<R>();
```

```
    for(T t : seq)
      result.add(func.function(t));
    return result;
  }
  // Applica un predicato unario a ogni elemento della
  // sequenza e restituisce un elenco di elementi che
restituiscono "true":
  public static <T> List<T>
  filter(Iterable<T> seq, UnaryPredicate<T> pred) {
    List<T> result = new ArrayList<T>();
    for(T t : seq)
      if(pred.test(t))
        result.add(t);
    return result;
  }
  // Per utilizzare i metodi generici precedenti, dovete
  // creare degli oggetti funzione che adatterete alle vostre
  // esigenze:
  static class IntegerAdder implements Combiner<Integer> {
    public Integer combine(Integer x, Integer y) {
      return x + y;
    }
  }
  static class
  IntegerSubtracter implements Combiner<Integer> {
    public Integer combine(Integer x, Integer y) {
      return x - y;
    }
  }
  static class
  BigDecimalAdder implements Combiner<BigDecimal> {
    public BigDecimal combine(BigDecimal x, BigDecimal y) {
      return x.add(y);
    }
  }
  static class
  BigIntegerAdder implements Combiner<BigInteger> {
```

```
    public BigInteger combine(BigInteger x, BigInteger y) {
      return x.add(y);
    }
  }
  static class
  AtomicLongAdder implements Combiner<AtomicLong> {
    public AtomicLong combine(AtomicLong x, AtomicLong y) {
      // Non e' chiaro se questo abbia un senso:
      return new AtomicLong(x.addAndGet(y.get()));
    }
  }
  // E' anche possibile realizzare una UnaryFunction di tipo
  // "ulp" (Units in the last place, unita' in ultima
  // posizione):
  static class BigDecimalUlp
  implements UnaryFunction<BigDecimal,BigDecimal> {
    public BigDecimal function(BigDecimal x) {
      return x.ulp();
    }
  }
  static class GreaterThan<T extends Comparable<T>>
  implements UnaryPredicate<T> {
    private T bound;
    public GreaterThan(T bound) { this.bound = bound; }
    public boolean test(T x) {
      return x.compareTo(bound) > 0;
    }
  }
  static class MultiplyingIntegerCollector
  implements Collector<Integer> {
    private Integer val = 1;
    public Integer function(Integer x) {
      val *= x;
      return val;
    }
    public Integer result() { return val; }
```

```
}
public static void main(String[] args) {
  // Generici, argomenti e boxing lavorano insieme:
  List<Integer> li = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
  Integer result = reduce(li, new IntegerAdder());
  print(result);

  result = reduce(li, new IntegerSubtracter());
  print(result);

  print(filter(li, new GreaterThan<Integer>(4)));

  print(forEach(li,
    new MultiplyingIntegerCollector()).result());

  print(forEach(filter(li, new GreaterThan<Integer>(4)),
    new MultiplyingIntegerCollector()).result());

  MathContext mc = new MathContext(7);
  List<BigDecimal> lbd = Arrays.asList(
    new BigDecimal(1.1, mc), new BigDecimal(2.2, mc),
    new BigDecimal(3.3, mc), new BigDecimal(4.4, mc));
  BigDecimal rbd = reduce(lbd, new BigDecimalAdder());
  print(rbd);

  print(filter(lbd,
    new GreaterThan<BigDecimal>(new BigDecimal(3))));

  // Usa la funzionalità di generazione di numeri primi
  // di BigInteger:
  List<BigInteger> lbi = new ArrayList<BigInteger>();
  BigInteger bi = BigInteger.valueOf(11);
  for(int i = 0; i < 11; i++) {
    lbi.add(bi);
    bi = bi.nextProbablePrime();
  }
```

```
      print(lbi);

      BigInteger rbi = reduce(lbi, new BigIntegerAdder());
      print(rbi);
      // La somma di questo elenco di numeri primi e' anch'essa
      // un numero primo:
      print(rbi.isProbablePrime(5));

      List<AtomicLong> lal = Arrays.asList(
        new AtomicLong(11), new AtomicLong(47),
        new AtomicLong(74), new AtomicLong(133));
      AtomicLong ral = reduce(lal, new AtomicLongAdder());
      print(ral);

      print(transform(lbd,new BigDecimalUlp()));
    }
  } /* Output:
  28
  -26
  [5, 6, 7]
  5040
  210
  11.000000
  [3.300000, 4.400000]
  [11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
  311
  true
  265
  [0.000001, 0.000001, 0.000001, 0.000001]
  *///:~
```

Iniziate definendo le interfacce per i diversi tipi di oggetti funzione, che l'autore ha creato sulla base delle sue esigenze via via che metteva a punto i vari metodi; la classe **Combiner** è stata invece suggerita da un corrispondente anonimo in uno degli articoli pubblicati su *www.mindview.net*. La classe **Combiner** permette di prescindere dai dettagli specifici derivanti dal tentativo di aggiungere questi due oggetti, limitandosi ad affermare che devono essere

combinati in qualche modo. Di conseguenza, è evidente che **IntegerAdder** e **IntegerSubtracter** possono essere tipi di **Combiner**.

L'oggetto **UnaryFunction** accetta un solo argomento e produce un risultato; tenete presente che argomento e risultato non devono essere necessariamente dello stesso tipo. Un **Collector** è utilizzato come "parametro di raccolta" (*collecting parameter*), che vi consente di estrarre i risultati finali. Un **UnaryPredicate** fornisce un risultato di tipo **boolean**. Possono essere definiti altri oggetti funzione, tuttavia questi sono sufficienti per fare il punto della situazione.

La classe **Functional** contiene un certo numero di metodi generici che applicano gli oggetti funzione alle sequenze; il metodo **reduce()** applica la funzione in un **Combiner** a ogni elemento di una sequenza, per fornire un unico risultato.

Il metodo **forEach()** accetta un **Collector** e applica la relativa funzione a ogni elemento, ignorando il risultato di ogni chiamata di funzione. Questa funzione può essere chiamata soltanto per il suo effetto collaterale (che, pur non rientrando in uno stile di programmazione "funzionale", è comunque utile), oppure **Collector** può mantenere il proprio stato interno per diventare un parametro di raccolta, come avviene in questo esempio.

Il metodo **transform()** produce un elenco chiamando un **UnaryFunction** su ogni oggetto nella sequenza e intercettandone il risultato.

Per concludere, **filter()** applica un **UnaryPredicate** a ogni oggetto della sequenza e registra i valori true in una **List**, che restituisce.

Potete definire altre funzioni generiche: la libreria STL di C++, per esempio, ne contiene un numero elevato. Il problema è stato risolto anche in alcune librerie open source come JGA (*Generic Algorithms for Java*).

In C++ la funzionalità latent typing si incarica di fare corrispondere le operazioni alle funzioni che vengono chiamate, ma in Java dovete scrivere gli oggetti funzione necessari per adattare i metodi generici alle vostre necessità particolari. Pertanto, la parte seguente della classe mostra le diverse implementazioni degli oggetti funzione: notate, per esempio, che **IntegerAdder** e **BigDecimalAdder** risolvono lo stesso problema (la somma di due numeri, rappresentati da due oggetti) chiamando le operazioni adatte per il tipo particolare. In questo modo si combinano i due pattern, Adapter e Strategy.

Nel metodo **main()** potete osservare che ogni chiamata di metodo passa una sequenza con l'oggetto funzione appropriato. Inoltre, avrete certamente notato come alcune espressioni possono diventare piuttosto complesse, per esempio:

```
forEach(filter(li, new GreaterThan(4)),
        new MultiplyingIntegerCollector()).result()
```

Questa riga di istruzione produce una lista selezionando tutti gli elementi che in **li** sono maggiori di 4, quindi applica il metodo **MultiplyingIntegerCollector()** all'elenco risultante ed estrae i risultati con **result()**. Non si è ritenuto opportuno entrare nel dettaglio della parte rimanente del codice, per offrirvi uno spunto di esercizio d'analisi.

**Esercizio 42** (5) Create due classi separate, che non abbiano niente in comune: ogni classe dovrà contenere un valore e almeno alcuni metodi che producono tale valore e lo sottopongono a una modifica. Cambiate **Functional.java** in modo che esegua operazioni funzionali sulle collezioni delle vostre classi: tenete presente che tali operazioni non dovranno essere aritmetiche come quelle in **Functional.java**.

## Riepilogo: il casting è davvero così sconveniente?

Avendo lavorato per spiegare ad altri i modelli C++ sin dall'inizio della sua carriera, probabilmente l'autore ha portato avanti la discussione su questo argomento più a lungo di chiunque altro. Solo di recente ha smesso di domandarsi quanto spesso questo argomento sia valido: quante reali probabilità esistono che il problema descritto dall'autore si verifichi?

La storia è questa. Uno dei punti più probabili in cui utilizzare un meccanismo di tipo generico è nelle classi contenitore come **List**, **Set**, **Map** ecc. che avete visto nel Volume 1, Capitolo 11 e che rivedrete in dettaglio nel Capitolo 5 di questo volume. Prima di Java SE5, quando inserivate un oggetto in un contenitore dovevate eseguirne l'upcast a **Object**, perdendo le informazioni sul tipo. Quando volevate estrarre questi oggetti per utilizzarli, dovevate anche eseguirne il downcast al tipo corretto. L'esempio presentato dall'autore si riferiva a una **List** di **Cat**; una variante che si serve di **Apple** e **Orange** è stata riportata all'inizio del Volume 1, Capitolo 11. Senza la versione generica dei contenitori introdotta da Java SE5 potevate inserire **Object** e ottenevate altri **Object**, quindi era possibile inserire un **Dog** in una **List** di **Cat**.

Tuttavia, le versioni di Java precedenti l'introduzione dei generici vi impedivano di utilizzare in modo *non corretto* gli oggetti inseriti nel contenitore. Se inserite un **Dog** in un contenitore di **Cat** e poi cercate di gestire tutti gli oggetti presenti nel contenitore come **Cat**, otterreste una **RuntimeException** al momento di estrarre il riferimento a **Dog** per cercare di convertirlo in **Cat**. Certamente scoprirete il problema, ma soltanto a runtime, non in fase di compilazione.

Nelle precedenti edizioni di questo manuale, l'autore ha affermato:

> *"Questa è più di una semplice seccatura, è qualcosa che può produrre dei bug difficili da individuare. Immaginate che una o più parti di un programma inseriscano oggetti in un contenitore: a seguito di un'eccezione potreste notare solo in una determinata parte del programma che un oggetto errato è stato inserito nel contenitore. A quel punto dovrete scoprire dove è avvenuto l'errato inserimento".*

L'autore, tuttavia, riesaminando ulteriormente l'argomento, ha iniziato a riflettere sulla questione. In primo luogo, quanto spesso si verifica? L'autore non ricorda che questo tipo di situazione gli sia mai accaduta, e quando ha chiesto conferme ai partecipanti a numerose conferenze, non ha mai sentito di nessuno cui fosse accaduto.

A prescindere da quanti controlli sul tipo offra Java, è sempre possibile scrivere programmi poco chiari e un programma scritto male, anche se compila, rimane sempre un programma redatto male. Forse la maggior parte dei programmatori utilizza contenitori con nomi sufficientemente descrittivi che, come "**cats**", forniscono un "indizio" visivo al programmatore che cercasse di aggiungervi un oggetto non **Cat**. E se anche avvenisse, per quanto tempo una cosa simile rimarrebbe nascosta? Si ha l'impressione che non appena si avviano test con dati reali, non ci voglia molto per vedere visualizzata un'eccezione.

Un altro autore ha persino affermato che un bug di questo tipo potrebbe rimanere nascosto per "anni e anni". Tuttavia, all'autore non risulta nessuna pioggia di segnalazioni di bug del tipo "cane in un elenco di gatti", né che simili bug siano avvenuti spesso. Considerato ciò che vedrete nel Volume 3, Capitolo 1 con i thread, è abbastanza facile e comune che si verifichino bug i quali possono comparire molto di rado e darvi soltanto una vaga indicazione di ciò che sta accadendo. Quindi, è davvero il problema del "cane in un elenco di gatti" la ragione per cui Java ha integrato questa caratteristica così importante e complessa?

L'autore ritiene che lo scopo dei cosiddetti "generici", in un linguaggio di utilizzo generale (non necessariamente nell'implementazione offerta da Java), sia l'espressività, non la semplice creazione di contenitori caratterizzati dalla sicurezza dei tipi. I "contenitori sicuri" sono un effetto secondario della capacità di generare codice di utilizzo più generale.

Quindi, sebbene l'argomento "cane in un elenco di gatti" venga spesso utilizzato per giustificare i generici, è discutibile e, come si è detto all'inizio del capitolo, l'autore non ritiene che sia questo il vero concetto di base dei generici. Al contrario, i generici sono ciò che indica il loro nome: un modo per scrive-

re codice più "generico", meno limitato dai tipi con cui può operare, per far sì che una singola porzione di codice possa essere applicata a più tipi.

Come avete visto in questo capitolo, è relativamente facile scrivere classi "contenitore" davvero generiche, come sono effettivamente i contenitori Java, ma scrivere codice generico che manipola i relativi tipi generici richiede uno sforzo supplementare, sia da parte del creatore del classe, sia da parte del consumatore della classe: entrambi devono capire il concetto e l'implementazione del design pattern Adapter. Questo impegno aggiuntivo riduce la facilità d'utilizzo di questa caratteristica e può così renderla meno applicabile in settori in cui invece fornirebbe un valore aggiunto.

Notate anche che, dal momento che i generici in Java sono stati introdotti in un secondo tempo e non implementati all'origine del linguaggio, alcuni contenitori non hanno potuto essere resi robusti quanto si sarebbe voluto. Considerate, per esempio, una **Map**, in particolare i metodi **containsKey(Object key)** e **get(Object key)**: se questa classe fosse stata progettata con i generici già implementati, tali metodi avrebbero utilizzato tipi parametrizzati anziché **Object**, permettendo così il controllo in fase di compilazione che i generici dovrebbero offrire. Nelle mappe di C++, per esempio, il tipo chiave viene sempre controllato al momento della compilazione.

Una cosa è molto chiara: introdurre qualsiasi genere di meccanismo generico in una versione successiva di un linguaggio, dopo che questo è entrato nell'utilizzo generale, è un'operazione che crea disordine, e non può avvenire senza inconvenienti. In C++ i template sono stati introdotti nella versione ISO iniziale del linguaggio, e persino questo ha cautilizzato disagi, in quanto esisteva una versione precedente priva di modelli, antecedente al primo C++ standard. In realtà, quindi, i template sono sempre stati parte di C++. In Java, invece, i generici sono stati introdotti quasi 10 anni dopo l'uscita della prima versione del linguaggio, pertanto i problemi di migrazione ai generici sono stati e continuano a essere considerevoli, con un impatto notevole sulla progettazione con i generici. Il risultato è che voi, come programmatori, dovrete sottostare ad alcuni disagi a causa della mancanza di lungimiranza dimostrata dai progettisti di Java quando hanno creato la versione 1.0 del linguaggio. Quando Java è stato concepito i progettisti naturalmente conoscevano i template C++, e hanno anche valutato la possibilità di includerli nel linguaggio. Hanno tuttavia deciso di ometterli, probabilmente perché incalzati dalla fretta: di conseguenza, sia il linguaggio sia i programmatori che lo utilizzano devono sopportare alcuni inconvenienti. Soltanto il tempo saprà dire quale impatto definitivo avrà sul linguaggio l'approccio di Java ai generici.

Alcuni linguaggi, in particolare Nice (*http://nice.sourceforge.net*), che genera bytecode Java e utilizza le librerie Java esistenti, e NextGen (*http://japan.cs.rice.edu/nextgen*), offrono un maggiore livello di "pulizia" e un approccio meno drastico ai tipi parametrizzati. Non è impossible ipotizzare che un linguaggio di questo tipo possa diventare il successore di Java, perché adotta esattamente lo stesso approccio che C++ ha adottato con C: servirsi di quello che esiste e migliorarlo.

## Ulteriori letture

Il documento introduttivo ufficiale sui generici è *Generics in the Java Programming Language*, di Gilad Bracha, scaricabile dall'indirizzo *http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf*.

Le FAQ di Angelika Langer sui generici di Java sono una risorsa molto utile, consultabile all'indirizzo *www.angelikalanger.com/GenericsFAQ/JavaGenericsFaq.html.*

Potrete approfondire l'argomento dei metacaratteri consultando l'articolo *Adding Wildcards to the Java Programming Language*, di Torgerson, Ernst, Hansen, von der Ahé, Bracha e Gafter, disponibile all'indirizzo *www.jot.fm/issues/issue_2004_12/article5*.

> *La soluzione degli esercizi è disponibile nel documento* The Thinking in Java Annotated Solution Guide*, in vendita all'indirizzo* www.mindview.net.

# Capitolo 4

# Array

Al termine del Volume 1, Capitolo 5 avete imparato come definire e inizializzare un array.

L'approccio più semplice da adottare nei confronti degli array consiste nel considerarne semplicemente la creazione, il popolamento, la selezione degli elementi ricorrendo a indici **int** ed evitando di modificarne il formato. Nella maggior parte dei casi queste conoscenze vi basteranno, ma a volte avrete la necessità di eseguire operazioni più specializzate e potreste anche considerare l'utilizzo di un array in alternativa a un contenitore, più flessibile. Questo capitolo vi mostrerà come considerare gli array da un punto di vista più profondo.

## Perché gli array sono speciali

Java offre diversi modi per contenere gli oggetti, quindi che cosa rende così speciali gli array?

Sono tre le caratteristiche che distinguono gli array da altri tipi di contenitori: efficienza, tipo e possibilità di contenere i primitivi. L'array è lo strumento Java più efficiente per registrare e accedere in modo casuale a una sequenza di riferimenti a oggetti. L'array è una semplice sequenza lineare che velocizza l'accesso ai sin-

goli elementi: la contropartita di questa velocità è costituita dal fatto che il formato di un oggetto array è fisso e non può essere cambiato per tutta la durata dell'array. Potreste pensare di utilizzare un **ArrayList** (si veda il Volume 1, Capitolo 11) che all'occorrenza assegna automaticamente nuovo spazio, creando un nuovo elenco e spostando tutti i riferimenti dal vecchio elenco a quello nuovo. Di norma dovreste quindi preferire un contenitore **ArrayList** a un array, ma questa flessibilità comporta alcuni oneri, dal momento che l'**ArrayList** è decisamente meno efficiente di un array.

Entrambi i componenti garantiscono che se ne faccia un utilizzo corretto e visualizzano una **RuntimeException** se ne eccedete i limiti, per indicare un errore di programmazione.

Prima della comparsa dei generici, le altre classi contenitore gestivano gli oggetti come se non avessero alcun tipo specifico: in pratica li consideravano di tipo **Object**, la classe radice di tutte le classi Java. In questo senso gli array sono migliori dei contenitori nella versione precedente ai generici, perché un array viene creato per contenere un tipo specifico: questo significa che il compilatore esegue un controllo di tipo per impedirvi di inserire o di estrarre il tipo errato. Naturalmente Java vi impedirà di trasmettere un messaggio inadeguato a un oggetto, sia in fase di compilazione sia al momento dell'esecuzione, pertanto in un modo o nell'altro questo non comporta rischi: è bene che il compilatore evidenzi questi avvertimenti ed è meno probabile che il programmatore utente rimanga sorpreso da un'eccezione.

Un array può contenere primitivi, mentre non può farlo un contenitore nella versione precedente ai generici. Con i generici, tuttavia, i contenitori possono specificare e controllare il tipo di oggetti che contengono, e grazie all'autoboxing comportarsi come se fossero in grado di contenere i primitivi, dal momento che la conversione avviene automaticamente. Ecco un esempio che paragona gli array ai contenitori generici:

```
//: arrays/ContainerComparison.java
import java.util.*;
import static net.mindview.util.Print.*;

class BerylliumSphere {
  private static long counter;
  private final long id = counter++;
  public String toString() { return "Sphere" + id; }
}
```

```
public class ContainerComparison {
  public static void main(String[] args) {
    BerylliumSphere[] spheres = new BerylliumSphere[10];
    for(int i = 0; i < 5; i++)
      spheres[i] = new BerylliumSphere();
    print(Arrays.toString(spheres));
    print(spheres[4]);

    List<BerylliumSphere> sphereList =
      new ArrayList<BerylliumSphere>();
    for(int i = 0; i < 5; i++)
      sphereList.add(new BerylliumSphere());
    print(sphereList);
    print(sphereList.get(4));

    int[] integers = { 0, 1, 2, 3, 4, 5 };
    print(Arrays.toString(integers));
    print(integers[4]);

    List<Integer> intList = new ArrayList<Integer>(
      Arrays.asList(0, 1, 2, 3, 4, 5));
    intList.add(97);
    print(intList);
    print(intList.get(4));
  }
} /* Output:
[Sphere 0, Sphere 1, Sphere 2, Sphere 3, Sphere 4, null,
null, null, null, null]
Sphere 4
[Sphere 5, Sphere 6, Sphere 7, Sphere 8, Sphere 9]
Sphere 9
[0, 1, 2, 3, 4, 5]
4
[0, 1, 2, 3, 4, 5, 97]
4
*///:~
```

Entrambi gli strumenti per la memorizzazione degli oggetti sono controllati per quanto riguarda il tipo, e l'unica differenza apparente è che l'array si serve di **[]** per accedere agli elementi, mentre una **List** ricorre a metodi quali **add()** e **get()**. La somiglianza tra array e l'**ArrayList** è intenzionale, in modo che sia concettualmente più facile passare da uno all'altro; in ogni caso, come avete visto nel Volume 1, Capitolo 11, i contenitori offrono molte più funzionalità rispetto agli array.

Grazie all'autoboxing, i contenitori sono diventati facili da utilizzare con i primitivi quasi quanto gli array: l'unico vantaggio rimasto agli array è quindi l'efficienza. Tuttavia, nel caso in cui si debba risolvere un problema più generale, l'array potrebbe rivelarsi troppo restrittivo, pertanto è consigliabile servirsi di una classe contenitore.

## Gli array sono oggetti di prima classe

A prescindere dal tipo di array con il quale intendete operare, l'identificativo di array non è altro che un riferimento a un vero oggetto generato in heap. Questo oggetto contiene i riferimenti ad altri oggetti, e può essere generato implicitamente come parte della sintassi di inizializzazione dell'array, oppure in modo esplicito con la parola chiave **new**. Una parte dell'oggetto array0 (l'unico campo o metodo cui possiate accedere) è il membro di sola lettura **length**, che indica quanti elementi può contenere l'array. La sintassi "**[]**" rappresenta l'unica altra modalità di accesso ammessa per gli array.

Il seguente esempio riepiloga le diverse tecniche di inizializzazione di un array e i modi in cui i riferimenti degli array possono essere assegnati ai diversi oggetti array. Il codice evidenzia anche che le tecniche d'utilizzo degli array di oggetti e degli array di primitivi sono quasi identiche: l'unica differenza è che i primi conservano i riferimenti, mentre gli array di primitivi memorizzano direttamente i valori primitivi.

```
//: arrays/ArrayOptions.java
// Inizializzazione e riassegnazione di array.
import java.util.*;
import static net.mindview.util.Print.*;

public class ArrayOptions {
  public static void main(String[] args) {
    // Array di oggetti:
    BerylliumSphere[] a; // Variabile locale non inizializzata
```

```
BerylliumSphere[] b = new BerylliumSphere[5];
// I riferimenti interni dell'array sono automaticamente
// inizializzati a null:
print("b: " + Arrays.toString(b));
BerylliumSphere[] c = new BerylliumSphere[4];
for(int i = 0; i < c.length; i++)
  if(c[i] == null) // Puo' controllare i riferimenti null
    c[i] = new BerylliumSphere();
// Inizializzazione aggregata:
BerylliumSphere[] d = { new BerylliumSphere(),
  new BerylliumSphere(), new BerylliumSphere()
};
// Inizializzazione aggregata dinamica:
a = new BerylliumSphere[]{
  new BerylliumSphere(), new BerylliumSphere(),
};
// (Nell'istruzione precedente, le virgole finali
// sono opzionali in entrambi i casi)
print("a.length = " + a.length);
print("b.length = " + b.length);
print("c.length = " + c.length);
print("d.length = " + d.length);
a = d;
print("a.length = " + a.length);

// Array di primitivi:
int[] e; // Riferimento null
int[] f = new int[5];
// I primitivi interni dell'array sono automaticamente
// inizializzati a zero:
print("f: " + Arrays.toString(f));
int[] g = new int[4];
for(int i = 0; i < g.length; i++)
  g[i] = i*i;
int[] h = { 11, 47, 93 };
// Errore di compilazione: variable e not initialized:
```

```
        //!print("e.length = " + e.length);
        print("f.length = " + f.length);
        print("g.length = " + g.length);
        print("h.length = " + h.length);
        e = h;
        print("e.length = " + e.length);
        e = new int[]{ 1, 2 };
        print("e.length = " + e.length);
    }
} /* Output:
b: [null, null, null, null, null]
a.length = 2
b.length = 5
c.length = 4
d.length = 3
a.length = 3
f: [0, 0, 0, 0, 0]
f.length = 5
g.length = 4
h.length = 3
e.length = 3
e.length = 2
*///:~
```

L'array **a** è una variabile locale non inizializzata e, finché non lo avrete inizializzato correttamente, il compilatore vi impedirà di eseguire qualsiasi cosa con questo riferimento. L'array **b** viene inizializzato sotto forma di un array di riferimenti a **BerylliumSphere**, ma nessun oggetto reale di tipo **BerylliumSphere** viene disposto in quell'array; comunque sia, potete sempre interrogare le dimensioni dell'array, dal momento che **b** punta a un oggetto legittimo. Questa tecnica richiede però attenzione: l'interrogazione di **length** consente di scoprire quanti elementi *possono* essere disposti nell'array, non quanti ve ne sono *effettivamente* contenuti. In ogni caso, quando create un oggetto array i suoi riferimenti vengono automaticamente inizializzati come **null**, di conseguenza per vedere se una specifica posizione dell'array contiene un oggetto vi basta controllare se essa equivale a **null**. In modo analogo, un array di primitivi viene inizializzato automaticamente a zero per i tipi numerici, a **(char)0** per i **char** e **false** per il tipo **boolean**.

L'array **c** mostra la creazione dell'oggetto array, seguita dall'assegnazione degli oggetti **BerylliumSphere** a tutti gli elementi che compongono l'array. L'array **d** illustra la sintassi di "inizializzazione aggregata", per mezzo della quale l'oggetto array viene creato (implicitamente con **new** nell'heap, come per l'array **c**) *e* nello stesso tempo inizializzato con oggetti **BerylliumSphere**, tutto nella stessa istruzione.

L'inizializzazione dell'array successivo può essere considerata come un'"inizializzazione aggregata dinamica". L'inizializzazione aggregata dell'array **d** deve essere utilizzata nel punto in cui **d** viene definito, mentre con la seconda sintassi potete generare e inizializzare ovunque un oggetto array. Per esempio, se **hide()** fosse un metodo che accetta un array di oggetti **BerylliumSphere** potreste chiamarlo

```
hide(d);
```

ma anche creare dinamicamente l'array da passare come argomento:

```
hide(new BerylliumSphere[]{ new BerylliumSphere(),
  new BerylliumSphere() });
```

In molti casi, questa sintassi rappresenta un modo più pratico di scrivere il codice.

L'espressione:

```
a = d;
```

evidenzia come sia possibile prendere un riferimento collegato a un oggetto array e assegnarlo a un altro oggetto array, esattamente come fareste con qualunque altro tipo di riferimento a oggetti. A questo punto, sia **a** sia **d** punteranno al medesimo oggetto array presente nell'heap.

La seconda parte dell'esempio **ArrayOptions.java** mostra che gli array contenenti tipi primitivi funzionano come quelli di oggetti: la sola differenza è che i primi contengono direttamente il valore primitivo, mentre gli array di oggetti contengono soltanto riferimenti a oggetti.

> **Esercizio 1** (2) Create un metodo che accetti un array di **BerylliumSphere** come argomento, poi chiamate il metodo creando l'argomento in modo dinamico. Dimostrate che la normale inizializzazione aggregata dell'array in questo caso non funziona, quindi scoprite le uniche situazioni in cui questa inizializzazione funziona e l'inizializzazione aggregata dinamica è ridondante.

## Restituzione dell'array

Supponete di scrivere un programma contenente un metodo che non restituisce un oggetto, ma un intero gruppo di oggetti. In linguaggi come C e C++ un'operazione di questo tipo è complessa, poiché non è permesso restituire un array ma unicamente un puntatore a un array. Tale limitazione provoca problemi perché diventa difficile controllare il ciclo di vita dell'array, e questo spesso porta al verificarsi di perdite di memoria (*memory leak*).

In Java, invece, è sufficiente restituire l'array: non è necessario preoccuparsi delle responsabilità per questo array, che sarà presente finché ne avrete bisogno e verrà preso in carico dal garbage collector al momento opportuno.

Supponete, per esempio, di restituire un array di **String**:

```
//: arrays/IceCream.java
// Restituzione degli array dai metodi.
import java.util.*;

public class IceCream {
  private static Random rand = new Random(47);
  static final String[] FLAVORS = {
    "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
    "Mint Chip", "Moka Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie"
  };
  public static String[] flavorSet(int n) {
    if(n > FLAVORS.length)
      throw new IllegalArgumentException("Set too big");
    String[] results = new String[n];
    boolean[] picked = new boolean[FLAVORS.length];
    for(int i = 0; i < n; i++) {
      int t;
      do
        t = rand.nextInt(FLAVORS.length);
      while(picked[t]);
      results[i] = FLAVORS[t];
      picked[t] = true;
    }
```

```
    return results;
  }
  public static void main(String[] args) {
    for(int i = 0; i < 7; i++)
      System.out.println(Arrays.toString(flavorSet(3)));
  }
} /* Output:
[Rum Raisin, Mint Chip, Mocha Almond Fudge]
[Chocolate, Strawberry, Mocha Almond Fudge]
[Strawberry, Mint Chip, Mocha Almond Fudge]
[Rum Raisin, Vanilla Fudge Swirl, Mud Pie]
[Vanilla Fudge Swirl, Chocolate, Mocha Almond Fudge]
[Praline Cream, Strawberry, Mocha Almond Fudge]
[Mocha Almond Fudge, Strawberry, Mint Chip]
*///:~
```

Il metodo **flavorSet()** crea un array di **String** chiamato **results**, la dimensione del quale è **n**, determinata dall'argomento che viene passato al metodo. Il codice prosegue con la scelta casuale di sapori di gelato dall'array **FLAVORS** e il loro inserimento in **results**, che viene restituito. La restituzione di un array funziona come quella di qualsiasi altro oggetto: si tratta di un riferimento, per restituire il quale non è importante che l'array sia stato creato all'interno di **flavorSet()** o generato in qualunque altro punto. L'array rimarrà disponibile fino a quando vi servirà, poi il garbage collector si prenderà cura di ripulirlo quando non vi occorrerà più.

Come digressione, tenete presente che quando **flavorSet()** sceglie a caso i gusti di gelato si assicura anche che non sia già stato selezionato un sapore particolare. Questo compito è affidato a un ciclo **do**, in cui vengono eseguite scelte casuali finché non viene estratto un sapore che non sia già stato inserito nell'array **picked**: naturalmente sarebbe possibile anche eseguire un confronto di **String**, per valutare se la scelta casuale è già presente nell'array **results**. Se il ciclo restituisce un nuovo sapore, aggiunge la voce e cerca la successiva, quindi incrementa la variabile **i**.

Potete vedere dall'output che **flavorSet()** sceglie ogni volta i sapori in ordine casuale.

> **Esercizio 2** (1) Scrivete un metodo che accetti un argomento numerico di tipo **int** e restituisca un array della dimensione specificata da **int**, popolato con oggetti **BerylliumSphere**.

## Array multidimensionali

Non è difficile creare array multidimensionali; per un array di questo genere contenente tipi primitivi, includete ogni vettore dell'array in una coppia di parentesi graffe:

```
//: arrays/MultidimensionalPrimitiveArray.java
// Creazione di array multidimensionali.
import java.util.*;

public class MultidimensionalPrimitiveArray {
  public static void main(String[] args) {
    int[][] a = {
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
    System.out.println(Arrays.deepToString(a));
  }
} /* Output:
[[1, 2, 3], [4, 5, 6]]
*///:~
```

Ogni gruppo di parentesi graffe vi sposta al livello successivo dell'array.

Questo esempio si serve del metodo **Arrays.deepToString()** (introdotto in Java SE5) che trasforma gli array multidimensionali in **String**, come risulta evidente dall'output.

Potete anche allocare un array tridimensionale servendovi dell'operatore **new**, come mostra l'esempio seguente:

```
//: arrays/ThreeDWithNew.java
import java.util.*;

public class ThreeDWithNew {
  public static void main(String[] args) {
    // Array tridimensionale a dimensione fissa:
    int[][][] a = new int[2][2][4];
    System.out.println(Arrays.deepToString(a));
  }
```

```
} /* Output:
[[[0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0,
0]]]
*///:~
```

Notate che i valori primitivi nell'array vengono inizializzati automaticamente, se non fornite loro un valore iniziale esplicito. Gli array di oggetti sono inizializzati a **null**.

Ogni vettore negli array che compongono la matrice può essere di lunghezza arbitraria; questa caratteristica ne fa un *ragged array*, letteralmente "array imperfetto".

```
//: arrays/RaggedArray.java
import java.util.*;

public class RaggedArray {
  public static void main(String[] args) {
    Random rand = new Random(47);
    // Array tridimensionale con vettori di lunghezza
    // variabile:
    int[][][] a = new int[rand.nextInt(7)][][];
    for(int i = 0; i < a.length; i++) {
      a[i] = new int[rand.nextInt(5)][];
      for(int j = 0; j < a[i].length; j++)
        a[i][j] = new int[rand.nextInt(5)];
    }
    System.out.println(Arrays.deepToString(a));
  }
} /* Output:
[[], [[0], [0], [0, 0, 0, 0]], [[], [0, 0], [0, 0]], [[0,
0, 0], [0], [0, 0, 0, 0]], [[0, 0, 0], [0, 0, 0], [0], []],
[[0], [], [0]]]
*///:~
```

Il primo **new** genera un array il cui primo vettore ha una dimensione casuale, mentre tutti gli altri sono indeterminati. Il secondo **new** posto nel ciclo **for** popola tutti gli elementi, ma lascia il terzo indice indeterminato fino a quando non viene chiamato il terzo operatore **new**.

Potete trattare gli array di oggetti non primitivi in modo simile. Osservate come è possibile raggruppare molte espressioni **new** tra parentesi graffe:

```
//: arrays/MultidimensionalObjectArrays.java
import java.util.*;

public class MultidimensionalObjectArrays {
  public static void main(String[] args) {
    BerylliumSphere[][] spheres = {
      { new BerylliumSphere(), new BerylliumSphere() },
      { new BerylliumSphere(), new BerylliumSphere(),
        new BerylliumSphere(), new BerylliumSphere() },
      { new BerylliumSphere(), new BerylliumSphere(),
        new BerylliumSphere(), new BerylliumSphere(),
        new BerylliumSphere(), new BerylliumSphere(),
        new BerylliumSphere(), new BerylliumSphere() },
    };
    System.out.println(Arrays.deepToString(spheres));
  }
} /* Output:
[[Sphere 0, Sphere 1], [Sphere 2, v 3, Sphere 4, Sphere 5],
[Sphere 6, Sphere 7, Sphere 8, Sphere 9, Sphere 10, Sphere 11,
Sphere 12, Sphere 13]]
*///:~
```

Come potete vedere, **spheres** è un altro *ragged array* nel quale la lunghezza di ogni elenco di oggetti è differente.

Anche l'autoboxing funziona con gli inizializzatori degli array:

```
//: arrays/AutoboxingArrays.java
import java.util.*;

public class AutoboxingArrays {
  public static void main(String[] args) {
    Integer[][] a = { // Autoboxing:
      { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 },
      { 21, 22, 23, 24, 25, 26, 27, 28, 29, 30 },
```

```
      { 51, 52, 53, 54, 55, 56, 57, 58, 59, 60 },
      { 71, 72, 73, 74, 75, 76, 77, 78, 79, 80 },
    };
    System.out.println(Arrays.deepToString(a));
  }
} /* Output:
[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [21, 22, 23, 24, 25, 26,
27, 28, 29, 30], [51, 52, 53, 54, 55, 56, 57, 58, 59, 60],
[71, 72, 73, 74, 75, 76, 77, 78, 79, 80]]
*///:~
```

Di seguito è mostrato come assemblare pezzo per pezzo un array di oggetti non primitivi:

```
//: arrays/AssemblingMultidimensionalArrays.java
// Creazione di array multidimensionali.
import java.util.*;

public class AssemblingMultidimensionalArrays {
  public static void main(String[] args) {
    Integer[][] a;
    a = new Integer[3][];
    for(int i = 0; i < a.length; i++) {
      a[i] = new Integer[3];
      for(int j = 0; j < a[i].length; j++)
        a[i][j] = i * j; // Autoboxing
    }
    System.out.println(Arrays.deepToString(a));
  }
} /* Output:
[[0, 0, 0], [0, 1, 2], [0, 2, 4]]
*///:~
```

L'operazione **i*j** è stata inserita soltanto per assegnare un valore significativo a **Integer**.

Il metodo **Arrays.deepToString()** funziona sia con gli array di tipi primitivi sia con quelli di oggetti:

```
//: arrays/MultiDimWrapperArray.java
// Array multidimensionali di oggetti "wrapper".
import java.util.*;

public class MultiDimWrapperArray {
  public static void main(String[] args) {
    Integer[][] a1 = { // Autoboxing
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
    Double[][][] a2 = { // Autoboxing
      { { 1.1, 2.2 }, { 3.3, 4.4 } },
      { { 5.5, 6.6 }, { 7.7, 8.8 } },
      { { 9.9, 1.2 }, { 2.3, 3.4 } },
    };
    String[][] a3 = {
      { "The", "Quick", "Sly", "Fox" },
      { "Jumped", "Over" },
      { "The", "Lazy", "Brown", "Dog", "and", "friend" },
    };
    System.out.println("a1: " + Arrays.deepToString(a1));
    System.out.println("a2: " + Arrays.deepToString(a2));
    System.out.println("a3: " + Arrays.deepToString(a3));
  }
} /* Output:
a1: [[1, 2, 3], [4, 5, 6]]
a2: [[[1.1, 2.2], [3.3, 4.4]], [[5.5, 6.6], [7.7, 8.8]],
[[9.9, 1.2], [2.3, 3.4]]]
a3: [[The, Quick, Sly, Fox], [Jumped, Over], [The, Lazy,
Brown, Dog, and, friend]]
*///:~
```

Anche in questo caso, negli array **Integer** e **Double** l'autoboxing di Java SE5 crea gli oggetti wrapper in modo automatico.

**Esercizio 3** (4) Scrivete un metodo che crei e inizializzi un array bidimensionale di **double**. Le dimensioni dell'array sono determinate dall'argomento del metodo e i valori di inizializzazione sono un ambito definito dai valori iniziale e finale forniti come argomenti al metodo. Generate un secondo metodo che visualizzi l'array generato dal primo. Nel metodo **main()** testate i due metodi, creando e visualizzando array di varie dimensioni.

**Esercizio 4** (2) Ripetete l'esercizio precedente con array tridimensionali.

**Esercizio 5** (1) Dimostrate che gli array multidimensionali di tipi non primitivi vengono inizializzati automaticamente a **null**.

**Esercizio 6** (1) Scrivete un metodo che accetti due argomenti **int**, indicanti le due dimensioni di un array bidimensionale. Il metodo dovrà creare e popolare un array bidimensionale con oggetti **BerylliumSphere** secondo le dimensioni stabilite dagli argomenti.

**Esercizio 7** (1) Ripetete l'esercizio precedente con un array tridimensionale.

## Array e generici

In generale, array e generici non vanno d'accordo; non è infatti possibile istanziare array di tipi parametrizzati:

```
Peel<Banana>[] peels = new Peel<Banana>[10]; // Illegale
```

La cancellazione rimuove le informazioni sui tipi di parametro, mentre gli array devono conoscere il tipo esatto che contengono per far rispettare la sicurezza sui tipi.

Potete tuttavia parametrizzare il tipo dell'array stesso:

```
//: arrays/ParameterizedArrayType.java

class ClassParameter<T> {
  public T[] f(T[] arg) { return arg; }
}

class MethodParameter {
  public static <T> T[] f(T[] arg) { return arg; }
}
```

```
public class ParameterizedArrayType {
  public static void main(String[] args) {
    Integer[] ints = { 1, 2, 3, 4, 5 };
    Double[] doubles = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    Integer[] ints2 =
      new ClassParameter<Integer>().f(ints);
    Double[] doubles2 =
      new ClassParameter<Double>().f(doubles);
    ints2 = MethodParameter.f(ints);
    doubles2 = MethodParameter.f(doubles);
  }
} ///:~
```

Notate la praticità garantita dall'utilizzo di un metodo parametrizzato in luogo di una classe parametrizzata: non è necessario istanziare una classe con un parametro per ogni tipo diverso al quale occorra applicarla, e potete anche rendere **static** il metodo. Naturalmente non è sempre possibile optare per un metodo parametrizzato anziché una classe parametrizzata.

Come avrete modo di constatare, non è del tutto esatto affermare che è impossibile creare array di tipi generici. È vero che il compilatore non vi consentirà di istanziare un array di un tipo generico, però vi permetterà di creare un riferimento a un array di questo tipo. Per esempio:

```
List<String>[] ls;
```

Questa sintassi viene gestita dal compilatore senza problemi, e benché non possiate creare un oggetto array reale che supporti i generici, potete creare un array di tipo non generalizzato e convertirlo:

```
//: arrays/ArrayOfGenerics.java
// E' possibile creare array di generici.
import java.util.*;

public class ArrayOfGenerics {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    List<String>[] ls;
    List[] la = new List[10];
```

```
    ls = (List<String>[])la; // Avvertimento "unchecked"
    ls[0] = new ArrayList<String>();
    // Il controllo di compilazione produce un errore:
    //! ls[1] = new ArrayList<Integer>();

    // Il problema e' che List<String> e' un sottotipo di Object
    Object[] objects = ls; // Cosi' l'asseganzione e' OK
    // Compila e si esegue senza problemi:
    objects[1] = new ArrayList<Integer>();

    // Tuttavia, se le vostre esigenze sono immediate potete
    // creare un array di generici, sebbene con un
    // avvertimento di tipo "unchecked":
    List<BerylliumSphere>[] spheres =
      (List<BerylliumSphere>[])new List[10];
    for(int i = 0; i < spheres.length; i++)
      spheres[i] = new ArrayList<BerylliumSphere>();
  }
} ///:~
```

Una volta che disponete di un riferimento a una **List<String>[]**, noterete che Java esegue alcuni controlli in fase di compilazione. Il problema è che gli array sono covarianti, cosicché una **List<String>[]** è anche un **Object[]**: potete quindi basarvi su questa analogia per assegnare un **ArrayList<Integer>** nel vostro array, senza errori in sede di compilazione né in fase di esecuzione.

Se sapete di non dover eseguire un upcast e se le vostre necessità sono relativamente semplici, potrete creare un array di generici: questa operazione garantirà un minimo controllo di tipo in fase di compilazione. Ricordate, tuttavia, che un contenitore è sempre una scelta potenzialmente preferibile rispetto a un array di generici.

In generale, scoprirete che i generici sono efficaci nei limiti di una classe o di un metodo. Internamente, di solito la cancellazione rende inutilizzabili i generici, pertanto non vi è permesso creare, per esempio, un array di un tipo generico in un caso come questo:

```
//: arrays/ArrayOfGenericType.java
// Gli array di tipi generici non compileranno.
```

```
public class ArrayOfGenericType<T> {
  T[] array; // OK
  @SuppressWarnings("unchecked")
  public ArrayOfGenericType(int size) {
    //! array = new T[size]; // Illegale
    array = (T[])new Object[size]; // Avvertimento "unchecked"
  }
  // Illegale:
  //! public <U> U[] makeArray() { return new U[10]; }
} ///:~
```

Si ha nuovamente a che fare con la cancellazione: questo esempio tenta di creare array di tipi che sono stati cancellati, e che pertanto risultano ignoti. Notate che potete creare un array di **Object** ed eseguire il cast, ma senza l'annotazione **@SuppressWarnings** otterrete un avvertimento "unchecked" da parte del compilatore, perché in realtà l'array non contiene effettivamente il tipo **T**, né controlla questo tipo a runtime.

In altri termini, se generate un array **String[]** Java si assicurerà, in sede di compilazione e di esecuzione, che questo possa contenere soltanto oggetti di tipo **String**; tuttavia, se create un **Object[]**, l'array accetterà qualsiasi tipo di dato tranne quelli primitivi.

**Esercizio 8** (1) Dimostrate le affermazioni del paragrafo precedente.

**Esercizio 9** (3) Create le classi necessarie per l'esempio **Peel<Banana>** e dimostrate che il compilatore non esegue la compilazione. Risolvete il problema utilizzando un **ArrayList**.

**Esercizio 10** (2) Modificate **ArrayOfGenerics.java** per utilizzare i contenitori, anziché gli array, e dimostrate che è possibile eliminare gli avvertimenti visualizzati al momento della compilazione.

## Creazione dei dati di prova

Quando eseguite esperimenti con gli array e con i programmi in generale, avrete necessità di generare facilmente array popolati con dati di prova. Gli strumenti illustrati in questo paragrafo vi saranno utili per includere valori o oggetti in un array di test.

## *Arrays.fill()*

La classe **Arrays** della libreria standard di Java dispone di un metodo **fill()** piuttosto banale, che si limita a duplicare un unico valore in tutte le posizioni dell'array oppure, in caso di oggetti, a copiare lo stesso riferimento in ogni posizione. Ecco un esempio:

```
//: arrays/FillingArrays.java
// Utilizzo di Arrays.fill()
import java.util.*;
import static net.mindview.util.Print.*;

public class FillingArrays {
  public static void main(String[] args) {
    int size = 6;
    boolean[] a1 = new boolean[size];
    byte[] a2 = new byte[size];
    char[] a3 = new char[size];
    short[] a4 = new short[size];
    int[] a5 = new int[size];
    long[] a6 = new long[size];
    float[] a7 = new float[size];
    double[] a8 = new double[size];
    String[] a9 = new String[size];
    Arrays.fill(a1, true);
    print("a1 = " + Arrays.toString(a1));
    Arrays.fill(a2, (byte)11);
    print("a2 = " + Arrays.toString(a2));
    Arrays.fill(a3, 'x');
    print("a3 = " + Arrays.toString(a3));
    Arrays.fill(a4, (short)17);
    print("a4 = " + Arrays.toString(a4));
    Arrays.fill(a5, 19);
    print("a5 = " + Arrays.toString(a5));
    Arrays.fill(a6, 23);
    print("a6 = " + Arrays.toString(a6));
    Arrays.fill(a7, 29);
    print("a7 = " + Arrays.toString(a7));
```

```
    Arrays.fill(a8, 47);
    print("a8 = " + Arrays.toString(a8));
    Arrays.fill(a9, "Hello");
    print("a9 = " + Arrays.toString(a9));
    // Manipolazione degli intervalli:
    Arrays.fill(a9, 3, 5, "World");
    print("a9 = " + Arrays.toString(a9));
  }
} /* Output:
a1 = [true, true, true, true, true, true]
a2 = [11, 11, 11, 11, 11, 11]
a3 = [x, x, x, x, x, x]
a4 = [17, 17, 17, 17, 17, 17]
a5 = [19, 19, 19, 19, 19, 19]
a6 = [23, 23, 23, 23, 23, 23]
a7 = [29.0, 29.0, 29.0, 29.0, 29.0, 29.0]
a8 = [47.0, 47.0, 47.0, 47.0, 47.0, 47.0]
a9 = [Hello, Hello, Hello, Hello, Hello, Hello]
a9 = [Hello, Hello, Hello, World, World, Hello]
*///:~
```

Potete popolare l'intero array oppure, come mostrano le due ultime dichiarazioni, inserire un intervallo di elementi. In ogni caso, dal momento che **Arrays.fill()** può essere chiamato con un solo valore di dati, i risultati non saranno particolarmente utili.

## Generatori di dati

Per generare array di dati più interessanti e caratterizzati da maggiore flessibilità è preferibile ricorrere al concetto di **Generator**, introdotto nel Capitolo 3. Se vi servite di un generatore in un'utility, potrete produrre qualsiasi genere di dati a seconda del **Generator** scelto: questo è un esempio del modello *Strategy*, nel quale ogni generatore rappresenta una diversa strategia.[1]

1. In questo caso specifico, comunque, la definizione del tipo di pattern non è netta e potreste obiettare che il generatore rappresenta invece il modello *Command*. Secondo l'opinione dell'autore, tuttavia, lo scopo di questa operazione è senza dubbio quello di popolare un array, e dal momento che il generatore svolge una parte di questo compito dovrebbe essere considerato più orientato alla "strategia" che al "comando".

In questo paragrafo vedrete alcuni **Generator** e le semplici tecniche per definirne alcuni personalizzati.

In primo luogo, il seguente codice mostra un insieme di generatori di conteggio per tutti i wrapper dei tipi primitivi e per le **String**. Le classi del generatore sono nidificate all'interno della classe **CountingGenerator**, in modo che possiate utilizzare lo stesso nome dei tipi di oggetto generati; per esempio, un generatore che produce oggetti **Integer** potrebbe essere creato con l'espressione **new CountingGenerator.Integer()**:

```
//: net/mindview/util/CountingGenerator.java
// Implementazioni di semplici generatori.
package net.mindview.util;

public class CountingGenerator {
  public static class
  Boolean implements Generator<java.lang.Boolean> {
    private boolean value = false;
    public java.lang.Boolean next() {
      value = !value; // Si limita ad alternare valori true
                      // e false
      return value;
    }
  }
  public static class
  Byte implements Generator<java.lang.Byte> {
    private byte value = 0;
    public java.lang.Byte next() { return value++; }
  }
  static char[] chars = ("abcdefghijklmnopqrstuvwxyz" +
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ").toCharArray();
  public static class
  Character implements Generator<java.lang.Character> {
    int index = -1;
    public java.lang.Character next() {
      index = (index + 1) % chars.length;
      return chars[index];
    }
```

```
}
public static class
String implements Generator<java.lang.String> {
  private int length = 7;
  Generator<java.lang.Character> cg = new Character();
  public String() {}
  public String(int length) { this.length = length; }
  public java.lang.String next() {
    char[] buf = new char[length];
    for(int i = 0; i < length; i++)
      buf[i] = cg.next();
    return new java.lang.String(buf);
  }
}
public static class
Short implements Generator<java.lang.Short> {
  private short value = 0;
  public java.lang.Short next() { return value++; }
}
public static class
Integer implements Generator<java.lang.Integer> {
  private int value = 0;
  public java.lang.Integer next() { return value++; }
}
public static class
Long implements Generator<java.lang.Long> {
  private long value = 0;
  public java.lang.Long next() { return value++; }
}
public static class
Float implements Generator<java.lang.Float> {
  private float value = 0;
  public java.lang.Float next() {
    float result = value;
    value += 1.0;
    return result;
  }
```

```
    }
    public static class
    Double implements Generator<java.lang.Double> {
      private double value = 0.0;
      public java.lang.Double next() {
        double result = value;
        value += 1.0;
        return result;
      }
    }
  } ///:~
```

Ogni classe implementa un concetto diverso di "conteggio". Nel caso di **CountingGenerator.Character**, si tratta delle sole lettere maiuscole e minuscole ripetute più volte. La classe **CountingGenerator.String** utilizza **CountingGenerator.Character** per popolare un array di caratteri, poi trasformato in una **String**: le dimensioni dell'array sono determinate dall'argomento del costruttore. Notate che **CountingGenerator.String** si serve di un generatore **Generator<java.lang.Character>** di base, anziché un riferimento specifico a **CountingGenerator.Character**. In seguito, questo generatore potrà essere sostituito per produrre **RandomGenerator.String** in **RandomGenerator.java**.

Il codice seguente si riferisce a un'utility di test che si serve della riflessione con l'idioma **Generator** nidificato, per essere utilizzabile nei test di qualsiasi insieme di generatori che sia conforme a questa forma idiomatica:

```
  //: arrays/GeneratorsTest.java
  import net.mindview.util.*;

  public class GeneratorsTest {
    public static int size = 10;
    public static void test(Class<?> surroundingClass) {
      for(Class<?> type : surroundingClass.getClasses()) {
        System.out.print(type.getSimpleName() + ": ");
        try {
          Generator<?> g = (Generator<?>)type.newInstance();
          for(int i = 0; i < size; i++)
            System.out.printf(g.next() + " ");
          System.out.println();
```

```
      } catch(Exception e) {
        throw new RuntimeException(e);
      }
    }
  }
  public static void main(String[] args) {
    test(CountingGenerator.class);
  }
} /* Output:
Double: 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
Float: 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
Long: 0 1 2 3 4 5 6 7 8 9
Integer: 0 1 2 3 4 5 6 7 8 9
Short: 0 1 2 3 4 5 6 7 8 9
String: abcdefg hijklmn opqrstu vwxyzAB CDEFGHI JKLMNOP
QRSTUVW XYZabcd efghijk lmnopqr
Character: a b c d e f g h i j
Byte: 0 1 2 3 4 5 6 7 8 9
Boolean: true false true false true false true false true
false
*///:~
```

L'esempio prevede che la classe da testare contenga un insieme di oggetti **Generator** nidificati, ciascuno dei quali possiede un costruttore predefinito, cioè privo di argomenti. Il metodo di riflessione **getClasses()** crea tutte le classi nidificate, poi il metodo **test()** genera un'istanza di ogni generatore e visualizza il risultato fornito, chiamando dieci volte **next()**.

Nel codice seguente è presentato un insieme di **Generator** che utilizza il generatore di numeri casuali. Dal momento che il costruttore **Random** è inizializzato con un valore costante, l'output è ripetibile ogni volta che eseguite un programma il quale utilizza uno di questi generatori:

```
//: net/mindview/util/RandomGenerator.java
// Generatori che producono valori casuali.
package net.mindview.util;
import java.util.*;

public class RandomGenerator {
```

```
private static Random r = new Random(47);
public static class
Boolean implements Generator<java.lang.Boolean> {
  public java.lang.Boolean next() {
    return r.nextBoolean();
  }
}
public static class
Byte implements Generator<java.lang.Byte> {
  public java.lang.Byte next() {
    return (byte)r.nextInt();
  }
}
public static class
Character implements Generator<java.lang.Character> {
  public java.lang.Character next() {
    return CountingGenerator.chars[
      r.nextInt(CountingGenerator.chars.length)];
  }
}
public static class
String extends CountingGenerator.String {
  // Attiva il generatore casuale di Character:
  { cg = new Character(); } // Istanzia l'inizializzatore
  public String() {}
  public String(int length) { super(length); }
}
public static class
Short implements Generator<java.lang.Short> {
  public java.lang.Short next() {
    return (short)r.nextInt();
  }
}
public static class
Integer implements Generator<java.lang.Integer> {
  private int mod = 10000;
  public Integer() {}
```

```
    public Integer(int modulo) { mod = modulo; }
    public java.lang.Integer next() {
      return r.nextInt(mod);
    }
  }
  public static class
  Long implements Generator<java.lang.Long> {
    private int mod = 10000;
    public Long() {}
    public Long(int modulo) { mod = modulo; }
    public java.lang.Long next() {
      return new java.lang.Long(r.nextInt(mod));
    }
  }
  public static class
  Float implements Generator<java.lang.Float> {
    public java.lang.Float next() {
      // Crea un valore float con due cifre decimali:
      // produce un intero da 0 a 99
      int trimmed = Math.round(r.nextFloat() * 100);
      // converte in float e divide per 100
      return ((float)trimmed) / 100;
    }
  }
  public static class
  Double implements Generator<java.lang.Double> {
    public java.lang.Double next() {
      long trimmed = Math.round(r.nextDouble() * 100);
      return ((double)trimmed) / 100;
    }
  }
} ///:~
```

Osservate che **RandomGenerator.String** eredita da **CountingGenerator.String** e si limita ad attivare il nuovo generatore **Character**.

Per produrre numeri che non siano eccessivi, **RandomGenerator.Integer** è impostato a un modulo predefinito pari a 10.000, ma il costruttore sovracca-

rico vi permette di scegliere un valore inferiore; una tecnica analoga è utilizzata per **RandomGenerator.Long**. Nei generatori **Float** e **DoubleGenerator**, i valori dopo il separatore decimale vengono eliminati.

Ora potete riutilizzare **GeneratorsTest** per testare **RandomGenerator**:

```
//: arrays/RandomGeneratorsTest.java
import net.mindview.util.*;

public class RandomGeneratorsTest {
  public static void main(String[] args) {
    GeneratorsTest.test(RandomGenerator.class);
  }
} /* Output:
Double: 0.73 0.53 0.16 0.19 0.52 0.27 0.26 0.05 0.8 0.76
Float: 0.53 0.16 0.53 0.4 0.49 0.25 0.8 0.11 0.02 0.8
Long: 7674 8804 8950 7826 4322 896 8033 2984 2344 5810
Integer: 8303 3141 7138 6012 9966 8689 7185 6992 5746 3976
Short: 3358 20592 284 26791 12834 -8092 13656 29324 -1423
5327
String: bkInaMe sbtWHkj UrUkZPg wsqPzDy CyRFJQA HxxHvHq
XumcXZJ oogoYWM NvqeuTp nXsgqia
Character: x x E A J J m z M s
Byte: -60 -17 55 -14 -5 115 39 -37 79 115
Boolean: false true false false true true true true true
true
*///:~
```

Potete anche modificare il numero di valori prodotti cambiando il valore **GeneratorsTest.size**, che è **public**.

## *Creazione di array con i generatori*

Per utilizzare un generatore e produrre un array vi occorrono due strumenti di conversione, il primo dei quali utilizza qualsiasi **Generator** per creare un array di sottotipi **Object**. Per far fronte al problema dei primitivi, il secondo strumento accetta qualsiasi array di wrapper di tipi primitivi e crea l'array di primitivi relativo.

La prima utility ha due opzioni, rappresentate dal metodo **static** sovraccarico **array()**. La prima versione del metodo accetta un array esistente e lo popola per mezzo di un **Generator**, mentre la seconda versione accetta un oggetto **Class**, un **Generator** e il numero di elementi desiderato per generare un nuovo array, che popola anche in questo caso mediante un **Generator**. Tenete presente che l'utility produce soltanto array di sottotipi **Object** e non può generare array di primitivi:

```
//: net/mindview/util/Generated.java
package net.mindview.util;
import java.util.*;

public class Generated {
  // Popola un array esistente:
  public static <T> T[] array(T[] a, Generator<T> gen) {
    return new CollectionData<T>(gen, a.length).toArray(a);
  }
  // Crea un nuovo array:
  @SuppressWarnings("unchecked")
  public static <T> T[] array(Class<T> type,
      Generator<T> gen, int size) {
    T[] a =
      (T[])java.lang.reflect.Array.newInstance(type, size);
    return new CollectionData<T>(gen, size).toArray(a);
  }
} ///:~
```

La classe **CollectionData**, che verrà descritta in dettaglio nel Capitolo 5, genera un oggetto **Collection** popolato di elementi prodotti dal **Generator gen**. Il numero di elementi è determinato dal secondo argomento del costruttore. Tutti i sottotipi **Collection** mettono a disposizione un metodo **toArray()**, che popola l'array fornito come argomento con gli elementi ottenuti da **Collection**.

La seconda utility si serve della riflessione per creare dinamicamente un nuovo array del tipo e delle dimensioni adatti, che viene poi popolato utilizzando la stessa tecnica del primo metodo. Potete testare **Generated** ricorrendo a una delle classi **CountingGenerator** definite nel paragrafo precedente:

```
//: arrays/TestGenerated.java
import java.util.*;
```

```
import net.mindview.util.*;

public class TestGenerated {
  public static void main(String[] args) {
    Integer[] a = { 9, 8, 7, 6 };
    System.out.println(Arrays.toString(a));
    a = Generated.array(a,new CountingGenerator.Integer());
    System.out.println(Arrays.toString(a));
    Integer[] b = Generated.array(Integer.class,
        new CountingGenerator.Integer(), 15);
    System.out.println(Arrays.toString(b));
  }
} /* Output:
[9, 8, 7, 6]
[0, 1, 2, 3]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
*///:~
```

Anche se l'array **a** è inizializzato, i suoi valori vengono sovrascritti passandolo a **Generated.array()**, che sostituisce i valori ma lascia invariato l'array originale. L'inizializzazione di **b** mostra come creare da subito un array popolato.

Pur sapendo che i generici non funzionano con i primitivi, volete utilizzare i generatori per popolare degli array primitivi.

Per risolvere il problema, generate un convertitore che accetti qualsiasi array di oggetti wrapper e lo trasformi in un array dei tipi primitivi associati: senza questo strumento, dovreste creare generatori speciali per tutti i primitivi.

```
//: net/mindview/util/ConvertTo.java
package net.mindview.util;

public class ConvertTo {
  public static boolean[] primitive(Boolean[] in) {
    boolean[] result = new boolean[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i]; // Autounboxing
    return result;
  }
```

```
public static char[] primitive(Character[] in) {
  char[] result = new char[in.length];
  for(int i = 0; i < in.length; i++)
    result[i] = in[i];
  return result;
}
public static byte[] primitive(Byte[] in) {
  byte[] result = new byte[in.length];
  for(int i = 0; i < in.length; i++)
    result[i] = in[i];
  return result;
}
public static short[] primitive(Short[] in) {
  short[] result = new short[in.length];
  for(int i = 0; i < in.length; i++)
    result[i] = in[i];
  return result;
}
public static int[] primitive(Integer[] in) {
  int[] result = new int[in.length];
  for(int i = 0; i < in.length; i++)
    result[i] = in[i];
  return result;
}
public static long[] primitive(Long[] in) {
  long[] result = new long[in.length];
  for(int i = 0; i < in.length; i++)
    result[i] = in[i];
  return result;
}
public static float[] primitive(Float[] in) {
  float[] result = new float[in.length];
  for(int i = 0; i < in.length; i++)
    result[i] = in[i];
  return result;
}
```

```
  public static double[] primitive(Double[] in) {
    double[] result = new double[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
} ///:~
```

Ogni versione di **primitive()** crea un array primitivo adeguato e delle dimensioni corrette, poi copia gli elementi dall'array **in** dei tipi wrapper. Notate che l'autounboxing viene eseguito direttamente all'interno dell'espressione:

```
result[i] = in[i];
```

Ecco un esempio che mostra come utilizzare **ConvertTo** con entrambe le versioni di **Generated.array()**:

```
//: arrays/PrimitiveConversionDemonstration.java
import java.util.*;
import net.mindview.util.*;

public class PrimitiveConversionDemonstration {
  public static void main(String[] args) {
    Integer[] a = Generated.array(Integer.class,
        new CountingGenerator.Integer(), 15);
    int[] b = ConvertTo.primitive(a);
    System.out.println(Arrays.toString(b));
    boolean[] c = ConvertTo.primitive(
      Generated.array(Boolean.class,
        new CountingGenerator.Boolean(), 7));
    System.out.println(Arrays.toString(c));
  }
} /* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
[true, false, true, false, true, false, true]
*///:~
```

Per concludere, il programma seguente testa le utility per la generazione degli array ricorrendo alle classi **RandomGenerator**, e si assicura che ogni versione di **ConvertTo.primitive()** funzioni correttamente:

```
//: arrays/TestArrayGeneration.java
// Test delle utility che usano i generatori per popolare
// gli array.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class TestArrayGeneration {
  public static void main(String[] args) {
    int size = 6;
    boolean[] a1 = ConvertTo.primitive(Generated.array(
      Boolean.class, new RandomGenerator.Boolean(), size));
    print("a1 = " + Arrays.toString(a1));
    byte[] a2 = ConvertTo.primitive(Generated.array(
      Byte.class, new RandomGenerator.Byte(), size));
    print("a2 = " + Arrays.toString(a2));
    char[] a3 = ConvertTo.primitive(Generated.array(
      Character.class,
      new RandomGenerator.Character(), size));
    print("a3 = " + Arrays.toString(a3));
    short[] a4 = ConvertTo.primitive(Generated.array(
      Short.class, new RandomGenerator.Short(), size));
    print("a4 = " + Arrays.toString(a4));
    int[] a5 = ConvertTo.primitive(Generated.array(
      Integer.class, new RandomGenerator.Integer(), size));
    print("a5 = " + Arrays.toString(a5));
    long[] a6 = ConvertTo.primitive(Generated.array(
      Long.class, new RandomGenerator.Long(), size));
    print("a6 = " + Arrays.toString(a6));
    float[] a7 = ConvertTo.primitive(Generated.array(
      Float.class, new RandomGenerator.Float(), size));
    print("a7 = " + Arrays.toString(a7));
    double[] a8 = ConvertTo.primitive(Generated.array(
```

```
      Double.class, new RandomGenerator.Double(), size));
    print("a8 = " + Arrays.toString(a8));
  }
} /* Output:
a1 = [true, false, true, false, false, true]
a2 = [104, -79, -76, 126, 33, -64]
a3 = [Z, n, T, c, Q, r]
a4 = [-13408, 22612, 15401, 15161, -28466, -12603]
a5 = [7704, 7383, 7706, 575, 8410, 6342]
a6 = [7674, 8804, 8950, 7826, 4322, 896]
a7 = [0.01, 0.2, 0.4, 0.79, 0.27, 0.45]
a8 = [0.16, 0.87, 0.7, 0.66, 0.87, 0.59]
*///:~
```

**Esercizio 11** (2) Dimostrate che l'autoboxing non funziona con gli array.

**Esercizio 12** (1) Create un array inizializzato di valori **double** utilizzando **CountingGenerator** e visualizzate i risultati.

**Esercizio 13** (2) Popolate una **String** servendovi di **CountingGenerator.Character**.

**Esercizio 14** (6) Create un array per ogni tipo primitivo, quindi popolate tutti gli array utilizzando **CountingGenerator** e visualizzateli.

**Esercizio 15** (2) Modificate **ContainerComparison.java** realizzando un **Generator** di **BerylliumSphere** e modificate il metodo **main()** affinché utilizzi quel **Generator** con **Generated.array()**.

**Esercizio 16** (3) Basandovi su **CountingGenerator.java**, generate una classe **SkipGenerator** che produca nuovi valori incrementali in base a un argomento del costruttore. Modificate **TestArrayGeneration.java** per dimostrare che la vostra nuova classe funziona correttamente.

**Esercizio 17** (5) Create e testate un **Generator** di **BigDecimal** e assicuratevi che funzioni con i metodi di **Generated**.

## Utility per Arrays

In **java.util** è disponibile la classe **Arrays**, la quale raccoglie un insieme di metodi di utilità **static** per gli array. Esistono sei metodi di base: **equals()**, per valutare l'uguaglianza di due array (con il metodo **deepEquals()** per gli array multidimensionali); **fill()**, che avete già visto in precedenza nel capitolo;

**sort()**, per ordinare un array; **binarySearch()**, per trovare un elemento in un array ordinato; **toString()**, per produrre la rappresentazione **String** di un array; **hashCode()**, per produrre il valore hash di un array, che verrà esaminato nel Capitolo 5. Tutti questi metodi sono sovraccarichi per tutti i tipi primitivi e per **Object**. Inoltre, il metodo **Arrays.asList()** accetta qualunque sequenza o array e li trasforma in un contenitore **List**: questo metodo è gia stato analizzato nel Volume 1, Capitolo 11.

Prima di affrontare i diversi metodi di **Arrays**, occorre considerare un altro metodo molto utile che non appartiene a questa classe.

## *Copia di array*

La libreria standard Java fornisce un metodo **static**, **System.arraycopy()**, che è in grado di copiare gli array in modo molto più rapido rispetto all'utilizzo di un ciclo **for.**

Osservate questo esempio, che manipola gli array di **int**:

```
//: arrays/CopyingArrays.java
// Utilizzo di System.arraycopy()
import java.util.*;
import static net.mindview.util.Print.*;

public class CopyingArrays {
  public static void main(String[] args) {
    int[] i = new int[7];
    int[] j = new int[10];
    Arrays.fill(i, 47);
    Arrays.fill(j, 99);
    print("i = " + Arrays.toString(i));
    print("j = " + Arrays.toString(j));
    System.arraycopy(i, 0, j, 0, i.length);
    print("j = " + Arrays.toString(j));
    int[] k = new int[5];
    Arrays.fill(k, 103);
    System.arraycopy(i, 0, k, 0, k.length);
    print("k = " + Arrays.toString(k));
    Arrays.fill(k, 103);
    System.arraycopy(k, 0, i, 0, k.length);
```

```
    print("i = " + Arrays.toString(i));
    // Oggetti:
    Integer[] u = new Integer[10];
    Integer[] v = new Integer[5];
    Arrays.fill(u, new Integer(47));
    Arrays.fill(v, new Integer(99));
    print("u = " + Arrays.toString(u));
    print("v = " + Arrays.toString(v));
    System.arraycopy(v, 0, u, u.length/2, v.length);
    print("u = " + Arrays.toString(u));
  }
} /* Output:
i = [47, 47, 47, 47, 47, 47, 47]
j = [99, 99, 99, 99, 99, 99, 99, 99, 99, 99]
j = [47, 47, 47, 47, 47, 47, 47, 99, 99, 99]
k = [47, 47, 47, 47, 47]
i = [103, 103, 103, 103, 103, 47, 47]
u = [47, 47, 47, 47, 47, 47, 47, 47, 47, 47]
v = [99, 99, 99, 99, 99]
u = [47, 47, 47, 47, 47, 99, 99, 99, 99, 99]
*///:~
```

Gli argomenti di **arraycopy()** sono l'array di origine, la posizione nell'array di origine da cui iniziare a copiare, l'array di destinazione, la posizione nell'array di destinazione da cui iniziare la copia e il numero di elementi da copiare. Ovviamente, qualsiasi violazione dei limiti di array solleverà un'eccezione.

L'esempio mostra che sia gli array di primitivi sia quelli di oggetti possono essere copiati. Tuttavia, se copiate gli array di oggetti verranno copiati soltanto i relativi riferimenti, poiché anche in questo caso non è prevista la duplicazione degli oggetti stessi: questa tecnica, di cui troverete maggiori dettagli nei supplementi online di questo volume, è chiamata *copia shallow* (letteralmente, copia superficiale) o *copia byte-per-byte*.

Il metodo **System.arraycopy()** non eseguirà alcuna conversione, né di tipo autoboxing né di tipo autounboxing: i due array devono essere esattamente dello stesso tipo.

**Esercizio 18** (3) Create e popolate un array di **BerylliumSphere**, poi copiate questo array in uno nuovo e dimostrate che si tratta di una copia shallow.

## *Confronto di array*

La classe **Arrays** mette a disposizione il metodo **equals()** per valutare l'uguaglianza di interi array, che è sovraccarico per tutti i tipi primitivi e per **Object**. Per essere uguali, gli array devono avere lo stesso numero di elementi; l'equivalenza di ogni elemento di un array con il corrispondente elemento dell'altro array deve essere valutata utilizzando **equals()** per ogni coppia di elementi tra i due array. Tenete presente che per i primitivi viene utilizzato il metodo **equals()** della classe wrapper del primitivo corrispondente, per esempio **Integer.equals()** per il tipo **int**. Considerate questo esempio:

```
//: arrays/ComparingArrays.java
// Utilizzo di Arrays.equals()
import java.util.*;
import static net.mindview.util.Print.*;

public class ComparingArrays {
  public static void main(String[] args) {
    int[] a1 = new int[10];
    int[] a2 = new int[10];
    Arrays.fill(a1, 47);
    Arrays.fill(a2, 47);
    print(Arrays.equals(a1, a2));
    a2[3] = 11;
    print(Arrays.equals(a1, a2));
    String[] s1 = new String[4];
    Arrays.fill(s1, "Hi");
    String[] s2 = { new String("Hi"), new String("Hi"),
      new String("Hi"), new String("Hi") };
    print(Arrays.equals(s1, s2));
  }
} /* Output:
true
false
true
*///:~
```

In origine **a1** e **a2** sono esattamente uguali, pertanto l'output è "true", ma in seguito uno degli elementi viene modificato, rendendo il risultato "false". Nell'ultimo caso tutti gli elementi di **s1** puntano allo stesso oggetto, mentre **s2** ha cinque oggetti univoci; tuttavia l'uguaglianza di array è basata sui contenuti (tramite **Object.equals()**), pertanto il risultato è "true".

**Esercizio 19** (2) Generate una classe con un campo **int** inizializzato da un argomento del costruttore. Create due array di questi oggetti, utilizzando i valori di inizializzazione identici per ogni array, e mostrate che **Arrays.equals()** li considera diversi. Infine, aggiungete un metodo **equals()** alla vostra classe per risolvere il problema.

**Esercizio 20** (4) Dimostrate il funzionamento del metodo di utilità **deepEquals()** per gli array multidimensionali.

## *Confronto tra elementi di array*

L'ordinamento si ottiene per mezzo di una serie di confronti basati sul tipo effettivo dell'oggetto. Naturalmente una tecnica potrebbe essere quella di scrivere un metodo di ordinamento diverso per ogni tipo, ma tale codice non sarebbe riutilizzabile per eventuali nuovi tipi.

Come sapete, uno degli obiettivi primari di ogni progetto di programmazione è "separare ciò che cambia da quanto rimane invariato": in questo caso il codice che rimane lo stesso è l'algoritmo generale di ordinamento, mentre ciò che è soggetto a cambiamenti è il modo in cui vengono confrontati gli oggetti. Pertanto, anziché inserire il codice di confronto nelle diverse routine di ordinamento, è opportuno ricorrere al pattern *Strategy*.[2]

In questo modello Strategy la parte del codice che varia è incapsulata in una classe separata, l'oggetto Strategy: questo oggetto viene passato al codice "immutabile", che se ne serve per applicare i necessari algoritmi. In questo modo, potete realizzare diversi oggetti per esprimere modi diversi di confronto e passarli allo stesso codice di ordinamento.

La funzionalità di confronto in Java è supportata da due tecniche. La prima è quella di "confronto naturale", trasmessa a una determinata classe dall'implementazione dell'interfaccia **java.lang.Comparable**. Si tratta di un'interfaccia molto semplice composta di un solo metodo, **compareTo()**, il quale accetta come argomento un altro oggetto dello stesso tipo e restituisce un valore negativo se l'oggetto corrente è inferiore all'argomento, zero se equivale all'argomento e un valore positivo se l'oggetto corrente è maggiore dell'argomento.

2. *Design Patterns*, di Gamma, Helm, Johnson e Vlissides (Pearson Education Italia, 2002); si veda *Thinking in Patterns (with Java)*, disponibile all'indirizzo *http://mindview.net/Books/TIPatterns/*.

Di seguito, potete osservare una classe che implementa **Comparable** e dimostra la tecnica di confronto servendosi del metodo **Arrays.sort()** della libreria standard di Java:

```
//: arrays/CompType.java
// Implementazione di Comparable in una classe.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class CompType implements Comparable<CompType> {
  int i;
  int j;
  private static int count = 1;
  public CompType(int n1, int n2) {
    i = n1;
    j = n2;
  }
  public String toString() {
    String result = "[i = " + i + ", j = " + j + "]";
    if(count++ % 3 == 0)
      result += "\n";
    return result;
  }
  public int compareTo(CompType rv) {
    return (i < rv.i ? -1 : (i == rv.i ? 0 : 1));
  }
  private static Random r = new Random(47);
  public static Generator<CompType> generator() {
    return new Generator<CompType>() {
      public CompType next() {
        return new CompType(r.nextInt(100),r.nextInt(100));
      }
    };
  }
  public static void main(String[] args) {
    CompType[] a =
```

```
      Generated.array(new CompType[12], generator());
    print("before sorting:");
    print(Arrays.toString(a));
    Arrays.sort(a);
    print("after sorting:");
     print(Arrays.toString(a));
  }
} /* Output:
before sorting:
[[i = 58, j = 55], [i = 93, j = 61], [i = 61, j = 29]
, [i = 68, j = 0], [i = 22, j = 7], [i = 88, j = 28]
, [i = 51, j = 89], [i = 9, j = 78], [i = 98, j = 61]
, [i = 20, j = 58], [i = 16, j = 40], [i = 11, j = 22]
]
after sorting:
[[i = 9, j = 78], [i = 11, j = 22], [i = 16, j = 40]
, [i = 20, j = 58], [i = 22, j = 7], [i = 51, j = 89]
, [i = 58, j = 55], [i = 61, j = 29], [i = 68, j = 0]
, [i = 88, j = 28], [i = 93, j = 61], [i = 98, j = 61]
]
*///:~
```

Quando definite il metodo di confronto, avete la responsabilità di decidere che cosa significa confrontare uno dei vostri oggetti con un altro. In questo esempio il confronto utilizza soltanto i valori **i**, mentre i valori **j** vengono ignorati.

Il metodo **generator()** produce un oggetto che implementa l'interfaccia **Generator** creando una classe interna anonima: essa costruisce gli oggetti **CompType** inizializzandoli con valori casuali. Nel metodo **main()** il generatore è impiegato per popolare un array di **CompType**, che viene poi sottoposto a ordinamento. Se l'interfaccia **Comparable** non fosse stata implementata, chiamando il metodo **sort()** si produrrebbe una **ClassCastException** al momento dell'esecuzione, perché **sort()** esegue il cast del proprio argomento a **Comparable**.

Ora considerate la seconda tecnica. Supponete che qualcuno vi trasmetta una classe che non implementi l'interfaccia **Comparable**, o che la implementi in un modo che ritenete inadeguato per il tipo di confronto che volete applicare. In casi come questo potrete creare una classe separata che implementi

un'interfaccia denominata **Comparator**, già descritta brevemente nel Volume 1, Capitolo 11. Questo è un esempio del design pattern *Strategy*. L'interfaccia mette a disposizione due metodi, **compare()** ed **equals()**. Tuttavia non dovreste implementare **equals()** a meno di non dovere soddisfare particolari esigenze prestazionali, poiché ogni volta che creerete una classe questa erediterà implicitamente da **Object**, che pure ha un metodo **equals()**. Per soddisfare il contratto imposto dall'interfaccia, pertanto, vi basterà utilizzare il metodo **equals()** predefinito di **Object**.

La classe **Collection**, che vedrete in maggiore dettaglio nel prossimo capitolo, contiene un metodo **reverseOrder()** che produce un **Comparator** per invertire il criterio di ordinamento naturale. Tenete presente che il **Comparator** può essere applicato a **CompType**:

```
//: arrays/Reverse.java
// Il comparatore Collections.reverseOrder()
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class Reverse {
  public static void main(String[] args) {
    CompType[] a = Generated.array(
      new CompType[12], CompType.generator());
    print("before sorting:");
    print(Arrays.toString(a));
    Arrays.sort(a, Collections.reverseOrder());
    print("after sorting:");
    print(Arrays.toString(a));
  }
} /* Output:
before sorting:
[[i = 58, j = 55], [i = 93, j = 61], [i = 61, j = 29]
, [i = 68, j = 0], [i = 22, j = 7], [i = 88, j = 28]
, [i = 51, j = 89], [i = 9, j = 78], [i = 98, j = 61]
, [i = 20, j = 58], [i = 16, j = 40], [i = 11, j = 22]
]
after sorting:
[[i = 98, j = 61], [i = 93, j = 61], [i = 88, j = 28]
```

```
, [i = 68, j = 0], [i = 61, j = 29], [i = 58, j = 55]
, [i = 51, j = 89], [i = 22, j = 7], [i = 20, j = 58]
, [i = 16, j = 40], [i = 11, j = 22], [i = 9, j = 78]
]
*///:~
```

Potete anche realizzare un **Comparator** personalizzato; quello mostrato di seguito confronta gli oggetti **CompType** basandosi sui loro valori **j**, invece che sui valori **i**:

```
//: arrays/ComparatorTest.java
// Implementazione di un comparatore per una classe.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

class CompTypeComparator implements Comparator<CompType> {
  public int compare(CompType o1, CompType o2) {
    return (o1.j < o2.j ? -1 : (o1.j == o2.j ? 0 : 1));
  }
}

public class ComparatorTest {
  public static void main(String[] args) {
    CompType[] a = Generated.array(
      new CompType[12], CompType.generator());
    print("before sorting:");
    print(Arrays.toString(a));
    Arrays.sort(a, new CompTypeComparator());
    print("after sorting:");
    print(Arrays.toString(a));
  }
} /* Output:
before sorting:
[[i = 58, j = 55], [i = 93, j = 61], [i = 61, j = 29]
, [i = 68, j = 0], [i = 22, j = 7], [i = 88, j = 28]
, [i = 51, j = 89], [i = 9, j = 78], [i = 98, j = 61]
, [i = 20, j = 58], [i = 16, j = 40], [i = 11, j = 22]
```

```
]
after sorting:
[[i = 68, j = 0], [i = 22, j = 7], [i = 11, j = 22]
, [i = 88, j = 28], [i = 61, j = 29], [i = 16, j = 40]
, [i = 58, j = 55], [i = 20, j = 58], [i = 93, j = 61]
, [i = 98, j = 61], [i = 9, j = 78], [i = 51, j = 89]
]
*///:~
```

**Esercizio 21** (3) Provate a ordinare un array degli oggetti **BerylliumSphere** utilizzati nell'Esercizio 18 e implementate **Comparable** per risolvere il problema che vi si presenterà. Generate infine un **Comparator** per disporre gli oggetti in ordine inverso.

## *Ordinamento di array*

Con i metodi di ordinamento nativi potete ordinare qualsiasi array di primitivi, o qualunque array di oggetti che implementi **Comparable** o che sia associato a un **Comparator**.

Ecco un esempio che genera oggetti **String** casuali e li ordina:

```
//: arrays/StringSorting.java
// Ordinamento di un array di String.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class StringSorting {
  public static void main(String[] args) {
    String[] sa = Generated.array(new String[20],
      new RandomGenerator.String(5));
    print("Before sort: " + Arrays.toString(sa));
    Arrays.sort(sa);
    print("After sort: " + Arrays.toString(sa));
    Arrays.sort(sa, Collections.reverseOrder());
    print("Reverse sort: " + Arrays.toString(sa));
    Arrays.sort(sa, String.CASE_INSENSITIVE_ORDER);
    print("Case-insensitive sort: " + Arrays.toString(sa));
```

```
  }
} /* Output:
Before sort: [YNzbr, nyGcF, OWZnT, cQrGs, eGZMm, JMRoE,
suEcU, OneOE, dLsmw, HLGEa, hKcxr, EqUCB, bkIna, Mesbt,
WHkjU, rUkZP, gwsqP, zDyCy, RFJQA, HxxHv]
After sort: [EqUCB, HLGEa, HxxHv, JMRoE, Mesbt, OWZnT,
OneOE, RFJQA, WHkjU, YNzbr, bkIna, cQrGs, dLsmw, eGZMm,
gwsqP, hKcxr, nyGcF, rUkZP, suEcU, zDyCy]
Reverse sort: [zDyCy, suEcU, rUkZP, nyGcF, hKcxr, gwsqP,
eGZMm, dLsmw, cQrGs, bkIna, YNzbr, WHkjU, RFJQA, OneOE,
OWZnT, Mesbt, JMRoE, HxxHv, HLGEa, EqUCB]
Case-insensitive sort: [bkIna, cQrGs, dLsmw, eGZMm, EqUCB,
gwsqP, hKcxr, HLGEa, HxxHv, JMRoE, Mesbt, nyGcF, OneOE,
OWZnT, RFJQA, rUkZP, suEcU, WHkjU, YNzbr, zDyCy]
*///:~
```

Un particolare che non vi sarà sfuggito nell'output dell'algoritmo di **String** è il fatto che l'ordinamento è di tipo *lessicografico*, ovvero colloca per prime le parole con iniziale maiuscola, seguite dalle parole che iniziano con lettere minuscole. Se desiderate raggruppare tutte le parole a prescindere dall'iniziale minuscola o maiuscola, dovrete specificare il parametro **String.CASE_INSENSITIVE_ORDER**, come indicato nell'ultima chiamata a **sort()** dell'esempio.

L'algoritmo di ordinamento utilizzato nella libreria standard di Java è concepito per adattarsi in modo ottimale al tipo specifico da ordinare, *Quicksort* per i primitivi e un algoritmo di tipo *merge sort* per gli oggetti. In ogni caso non dovete preoccuparvi per le prestazioni, a meno che il vostro profiler non segnali un "collo di bottiglia" in corrispondenza del processo di ordinamento.

## *Ricerca in array ordinati*

Una volta che avete ordinato un array, potrete ricercare rapidamente un determinato elemento utilizzando **Arrays.binarySearch()**. Tuttavia, se provate a utilizzare **binarySearch()** su un array non ordinato, i risultati saranno imprevedibili. L'esempio seguente si serve di **RandomGenerator.Integer** per popolare un array, poi utilizza lo stesso generatore per produrre i valori di ricerca:

```
//: arrays/ArraySearching.java
// Utilizzo di Arrays.binarySearch().
```

```
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class ArraySearching {
  public static void main(String[] args) {
    Generator<Integer> gen =
      new RandomGenerator.Integer(1000);
    int[] a = ConvertTo.primitive(
      Generated.array(new Integer[25], gen));
    Arrays.sort(a);
    print("Sorted Array: " + Arrays.toString(a));
    while(true) {
      int r = gen.next();
      int location = Arrays.binarySearch(a, r);
      if(location >= 0) {
        print("Location of" + r + " is " + location +
          ", a[" + location + "] = " + a[location]);
        break; // Fuori del ciclo
      }
    }
  }
} /* Output:
Sorted Array: [128, 140, 200, 207, 258, 258, 278, 288, 322,
429, 511, 520, 522, 551, 555, 589, 693, 704, 809, 861, 861,
868, 916, 961, 998]
Location of 322 is 8, a[8] = 322
*///:~
```

Il ciclo **while** genera casualmente l'elemento da cercare; se il valore non viene trovato nell'array, il ciclo si ripete con un nuovo valore casuale.

**Arrays.binarySearch()** restituisce l'indice del valore trovato (maggiore o uguale a zero) se l'elemento richiesto viene trovato. In caso contrario (caso 1) il metodo restituisce un valore negativo, nel formato

```
-(insertion point) - 1
```

dove *insertion point* rappresenta il punto in cui dovrebbe essere inserito l'elemento se l'ordinamento fosse gestito manualmente, oppure (caso 2) restituisce **a.size()** se tutti gli elementi nell'array sono inferiori alla chiave specificata nella ricerca.

Se un array contiene elementi duplicati, non è possibile sapere a priori quale di essi verrà trovato, perché l'algoritmo di ricerca non è stato progettato per supportare gli elementi duplicati, sebbene li "tolleri". Se avete bisogno di una lista ordinata di elementi non duplicati, utilizzate un **TreeSet** per conservare l'ordinamento, oppure un **LinkedHashSet** per mantenere l'ordine di inserimento. Tali classi si occupano automaticamente di tutti i dettagli: soltanto nel caso in cui abbiate problemi di prestazioni potrete sostituire una di queste classi con un array gestito manualmente.

Potete ordinare un array di oggetti servendovi di un **Comparator**, che tuttavia non può essere utilizzato con gli array di tipi primitivi; lo stesso **Comparator** dovrà essere utilizzato quando eseguite il metodo **binarySearch()**.

Per esempio, ecco come modificare il programma **StringSorting.java** per eseguire una ricerca:

```
//: arrays/AlphabeticSearch.java
// Ricerca con un comparatore.
import java.util.*;
import net.mindview.util.*;

public class AlphabeticSearch {
  public static void main(String[] args) {
    String[] sa = Generated.array(new String[30],
      new RandomGenerator.String(5));
    Arrays.sort(sa, String.CASE_INSENSITIVE_ORDER);
    System.out.println(Arrays.toString(sa));
    int index = Arrays.binarySearch(sa, sa[10],
      String.CASE_INSENSITIVE_ORDER);
    System.out.println("Index: "+ index + "\n"+ sa[index]);
  }
} /* Output:
[bkIna, cQrGs, cXZJo, dLsmw, eGZMm, EqUCB, gwsqP, hKcxr,
HLGEa, HqXum, HxxHv, JMRoE, JmzMs, Mesbt, MNvqe, nyGcF,
ogoYW, OneOE, OWZnT, RFJQA, rUkZP, sgqia, slJrL, suEcU,
```

```
uTpnX, vpfFv, WHkjU, xxEAJ, YNzbr, zDyCy]
Index: 10
HxxHv
*///:~
```

**Comparator** deve essere passato al metodo **binarySearch()** come terzo argomento; in questo esempio la riuscita è garantita, poiché la voce da ricercare è selezionata dallo stesso array.

**Esercizio 22** (2) Dimostrate che i risultati dell'esecuzione di **binarySearch()** su un array non ordinato sono imprevedibili.

**Esercizio 23** (2) Create un array di **Integer** e popolatelo con valori **int** casuali utilizzando l'autoboxing, quindi disponetelo in ordine inverso ricorrendo a un **Comparator**.

**Esercizio 24** (3) Dimostrate che la classe dall'Esercizio 19 può essere oggetto di ricerca.

## Riepilogo

In questo capitolo avete visto che Java offre un discreto supporto agli array a dimensione fissa, di basso livello. Questo tipo di array privilegia le prestazioni rispetto alla flessibilità, come del resto fanno gli array in C++ e C. Nelle versioni iniziali di Java gli array di dimensione fissa a basso livello erano assolutamente necessari, non soltanto perché i progettisti avevano scelto di includere (anche per ragioni di prestazioni) i tipi primitivi, ma perché il supporto ai contenitori in quelle versioni era davvero minimo: quindi nelle prime versioni di Java era sempre opportuno ricorrere agli array.

Nelle versioni successive del linguaggio, però, il supporto ai contenitori è migliorato notevolmente, e oggi i contenitori tendono a superare gli array sotto molti aspetti, tranne che nelle prestazioni; occorre comunque dare atto che anche le prestazioni dei contenitori sono migliorate in modo considerevole. D'altro canto, come si è già fatto notare in altre occasioni, di solito le cause dei problemi di prestazioni non sono mai dove si ritiene che siano.

Con l'aggiunta dell'autoboxing e dei generici, l'inserimento di tipi primitivi nei contenitori è diventato molto semplice, e questo è un ulteriore motivo per sostituire i comuni array di basso livello con i contenitori. Inoltre, poiché i generici consentono di creare contenitori con funzionalità di sicurezza dei tipi, gli array hanno perso vantaggio anche da questo punto di vista.

Come avete visto in questo capitolo e come vedrete quando cercherete di utilizzarli, i generici sono piuttosto "ostili" nei confronti degli array: spesso, anche quando riuscirete a far convivere in qualche modo i generici e gli array (si veda il prossimo capitolo), vi ritroverete con avvertimenti "unchecked" durante la compilazione.

In molte occasioni i progettisti di Java hanno raccomandato all'autore di servirsi dei contenitori in luogo degli array, specialmente nella presentazione di esempi particolari; l'autore, in particolare, stava utilizzando gli array per dimostrare tecniche specifiche e quindi non aveva questa possibilità.

Tutte queste considerazioni indicano che, se programmate con versioni recenti di Java, dovreste privilegiare l'utilizzo dei contenitori rispetto a quello degli array, passando agli array soltanto quando le prestazioni dovessero rappresentare un elemento irrinunciabile, e se il ricorso agli array può essere di qualche beneficio.

Questa è una dichiarazione piuttosto forte, ma occorre considerare che alcuni linguaggi non offrono array di dimensioni fisse e a basso livello, disponendo solo di contenitori ridimensionabili con un maggior numero di funzionalità rispetto agli array di C, C++ e Java. Python (*www.python.org*), per esempio, possiede un tipo **list** che pur avendo la sintassi di base di un array ha un numero maggiore di funzioni, tra cui la possibilità di ereditare dallo stesso array:

```
#: arrays/PythonLists.py

aList = [1, 2, 3, 4, 5]
print type(aList) # <type 'list'>
print aList # [1, 2, 3, 4, 5]
print aList[4] # 5   Indicizzazione di base della list
aList.append(6) # Le list possono essere ridimensionate
aList += [7, 8] # Aggiunta di una list a un'altra list
print aList # [1, 2, 3, 4, 5, 6, 7, 8]
aSlice = aList[2:4]
print aSlice # [3, 4]

class MyList(list): # eredita da list
    # Definisce un metodo; il puntatore 'this' e' esplicito:
    def getReversed(self):
        reversed = self[:] # Copia di list usando lo slicing
```

```
        reversed.reverse() # Metodo nativo in list
        return reversed

list2 = MyList(aList) # Non occorre 'new' per creare l'oggetto
print type(list2) # <class '__main__.MyList'>
print list2.getReversed() # [8, 7, 6, 5, 4, 3, 2, 1]
#:~
```

La sintassi di base del linguaggio Python è stata rapidamente illustrata nel capitolo precedente. Questo codice crea una lista, semplicemente includendo tra parentesi quadre una sequenza di oggetti separati da virgole: il risultato è un oggetto con il tipo a runtime **list**; tenete presente che l'output dell'istruzione **print** è mostrato sotto forma di commento sulla stessa riga. Il risultato della visualizzazione di una **list** è lo stesso che si otterrebbe con il metodo **Arrays.toString()** in Java.

La generazione della sottosequenza di una **list** si ottiene per "slicing"[3], posizionando l'operatore ":" all'interno dell'operazione sull'indice. Oltre a questa, il tipo **list** dispone di molte altre funzionalità native.

**MyList** è una definizione di **class**; le classi di base sono incluse tra parentesi. All'interno della classe le istruzioni **def** definiscono i metodi, il primo argomento dei quali è automaticamente equivalente a **this** in Java, salvo il fatto che in Python è esplicito e per convenzione utilizza l'identificativo **self**, sebbene quest'ultima non sia una parola chiave. Notate che il costruttore viene ereditato in modo automatico.

Benché in Python tutto sia *effettivamente* un oggetto, inclusi i tipi interi e a virgola mobile, il linguaggio vi riserva una "scappatoia", permettendovi di ottimizzare porzioni critiche da un punto di vista prestazionale mediante la realizzazione di estensioni in C, C++ o tramite un'utility chiamata *Pyrex*, progettata per velocizzare le prestazioni del codice. In questo modo, viene garantita la purezza della programmazione OOP senza dover rinunciare a miglioramenti prestazionali.

---

3. Potente variante della normale indicizzazione, nella quale si scrivono l'indice iniziale e finale, separati da ":", di ciò che si vuole ottenere. Per esempio:

```
>>> disney=['pippo','pippo','pippo','pippo','pippo','minnie,'pippo','topolino',
'pippo']
>>> print disney[5:7]
['minnie','topolino']
```

Il linguaggio PHP 5 (*www.php.net*) rappresenta un'ulteriore evoluzione, dal momento che mette a disposizione un solo tipo di array, il quale si comporta sia come array indicizzato da interi (***int**-indexed array*) sia come array associativo, ossia una **Map**.

Dopo molti anni di evoluzione del linguaggio Java, è interessante domandarsi se i progettisti di Sun, supponendo di dover ricreare partendo da zero Java, deciderebbero di includere nel linguaggio i primitivi e gli array di basso livello. Se così non fosse, sarebbe possibile progettare un puro, vero linguaggio orientato agli oggetti: infatti, nonostante l'opinione corrente, Java non è un linguaggio OOP puro, proprio a causa di alcune "scorie" di basso livello. L'obiettivo di efficienza iniziale è sempre in primo piano, tuttavia nel corso degli anni si è potuto assistere a un'evoluzione che ha portato a un allontanamento da questo concetto, orientandosi sempre più verso l'utilizzo di componenti di livello più elevato quali i contenitori. A questo si aggiunge il fatto che se i contenitori potessero essere inseriti nel "nucleo del linguaggio", come avviene per altri linguaggi, il compilatore avrebbe opportunità nettamente migliori per ottimizzare il codice.

A parte queste considerazioni su "territori inesplorati", è innegabile che si dovrà rimanere ancorati agli array; li troverete spesso nel codice che avrete occasione di analizzare. Tenete presente, però, che i contenitori si rivelano quasi sempre la scelta migliore.

**Esercizio 25** (3) Riscrivete **PythonLists.py** in Java.

*La soluzione degli esercizi è disponibile nel documento* The Thinking in Java Annotated Solution Guide*, in vendita all'indirizzo* www.mindview.net.

# Capitolo 5

# Ancora sui contenitori

I concetti e le funzionalità di base della libreria dei contenitori Java sono stati presentati nel Volume 1, Capitolo 11, in misura sufficiente da permettervi di iniziare a utilizzare questi oggetti; in questo capitolo avrete modo di esplorare più a fondo questa importante libreria.

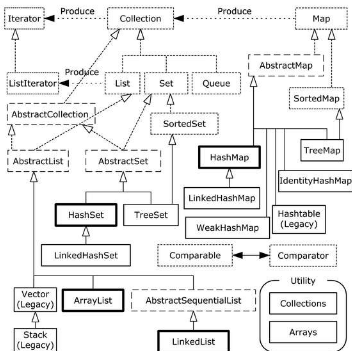
Per beneficiare delle funzionalità complete della libreria dei contenitori, è necessario disporre di maggiori informazioni di quante ne siano state fornite nel Volume 1, Capitolo 11; questo capitolo, tuttavia, richiede nozioni avanzate quali i generici, e per questo motivo si è ritenuto di collocarlo a questo punto del libro.

Dopo una presentazione più completa dei contenitori, vedrete come funziona il meccanismo di hashing e come scrivere metodi **hashCode()** ed **equals()** per lavorare con contenitori di tipo hashed, quindi comprenderete perché esistono versioni diverse di alcuni contenitori e come scegliere tra queste. Il capitolo si concluderà con l'esame di alcuni programmi di utilità di uso generico e classi speciali.

## Tassonomia completa dei contenitori

Nel Riepilogo del Volume 1, Capitolo 11 è stato presentato un diagramma semplificato della libreria dei contenitori Java; di seguito, la stessa libreria viene illu-

strata in modo più completo includendo le classi astratte e i vecchi componenti (*legacy*), con l'eccezione delle implementazioni di **Queue**:




**Tassonomia completa dei contenitori Java**

In Java SE5 sono stati aggiunti gli oggetti elencati di seguito.

1. L'interfaccia **Queue** (per implementare la quale, come si è detto nel Volume 1, Capitolo 11, è stata modificata **LinkedList**), e le sue implementazioni **PriorityQueue** e **BlockingQueue**, nelle varianti che vedrete nel Volume 3, Capitolo 1.
2. Un'interfaccia **ConcurrentMap** e la relativa implementazione **ConcurrentHashMap**, anch'esse utilizzate per i thread (si veda Volume 3, Capitolo 1).

3. **CopyOnWriteArrayList** e **CopyOnWriteArraySet**, anch'esse da impiegare nell'ambito della concorrenza (si veda il Volume 3, Capitolo 1).
4. **EnumSet** ed **EnumMap**, implementazioni speciali di **Set** e **Map** da utilizzare con **enum** (si veda il Volume 2, Capitolo 7).
5. Numerosi programmi di utilità nella classe **Collections**.

I riquadri con un tratteggio più lungo rappresentano le classi **abstract**: potete infatti notare alcune classi i cui nomi iniziano con "**Abstract**", le quali inizialmente potrebbero confondervi ma che in realtà sono semplici strumenti per implementare parzialmente una determinata interfaccia.

Se doveste realizzare un vostro **Set**, per esempio, non comincereste a implementare tutti i metodi dall'interfaccia **Set**; ereditereste invece da **AbstractSet** ed eseguireste gli interventi minimi necessari per costruire la nuova classe. In ogni caso, la libreria dei contenitori offre abbastanza funzionalità da soddisfare praticamente ogni vostra necessità, al punto che di norma potrete ignorare qualunque classe il cui nome cominci con "**Abstract**".

## Popolare i contenitori

Sebbene il problema della stampa e visualizzazione dei contenitori sia stato risolto in Java SE5, il compito di popolare di dati i contenitori risente degli stessi inconvenienti di **java.util.Arrays**. Come **Arrays**, anche la classe di supporto (*companion class*) chiamata **Collections** contiene metodi di utilità **static**, tra cui **fill()**; questo metodo, in modo analogo al suo corrispondente in **Arrays**, si limita a duplicare un solo riferimento di oggetto in tutto il contenitore. Inoltre, sebbene **fill()** funzioni soltanto per gli oggetti **List**, la lista risultante può essere passata a un costruttore o a un metodo **addAll()**:

```
//: containers/FillingLists.java
// I metodi Collections.fill() e Collections.nCopies().
import java.util.*;

class StringAddress {
  private String s;
  public StringAddress(String s) { this.s = s; }
  public String toString() {
    return super.toString() + " " + s;
  }
}
```

```
public class FillingLists {
  public static void main(String[] args) {
    List<StringAddress> list= new ArrayList<StringAddress>(
      Collections.nCopies(4, new StringAddress("Hello")));
    System.out.println(list);
    Collections.fill(list, new StringAddress("World!"));
    System.out.println(list);
  }
} /* Output:
[StringAddress@82ba41 Hello, StringAddress@82ba41 Hello,
StringAddress@82ba41 Hello, StringAddress@82ba41 Hello]
[StringAddress@923e30 World!, StringAddress@923e30 World!,
StringAddress@923e30 World!, StringAddress@923e30 World!]
*///:~
```

Questo esempio illustra due modi per riempire una **Collection** con riferimenti a un unico oggetto. Il primo, **Collections.nCopies()**, crea una **List** che viene passata al costruttore: questa tecnica popola **ArrayList**.

Il metodo **toString()** in **StringAddress** chiama **Object.toString()**, che produce il nome della classe seguito dalla rappresentazione esadecimale priva di segno del codice hash dell'oggetto (generato con il metodo **hashCode()**). L'output mostra che tutti i riferimenti sono impostati allo stesso oggetto, e questo è egualmente vero dopo la chiamata al secondo metodo, **Collections.fill()**. Il metodo **fill()** si rivela ancora meno utile per il fatto che può sostituire soltanto gli elementi che sono già nella **List**, senza aggiungerne di nuovi.

### *Una soluzione con Generator*

Praticamente qualsiasi sottotipo di **Collection** ha un costruttore che accetta un altro oggetto **Collection**, mediante il quale si può riempire il nuovo contenitore. Per semplificare la creazione di dati di prova, quindi, non dovete fare altro che costruire una classe la quale accetti come argomenti del costruttore un **Generator** (definito nel Capitolo 3 e ulteriormente analizzato nel Capitolo 4) e un valore **quantity**:

```
//: net/mindview/util/CollectionData.java
// Una Collection popolata utilizzando un oggetto Generator.
package net.mindview.util;
import java.util.*;
```

```
public class CollectionData<T> extends ArrayList<T> {
  public CollectionData(Generator<T> gen, int quantity) {
    for(int i = 0; i < quantity; i++)
      add(gen.next());
  }
  // Un metodo di comodo di uso generico:
  public static <T> CollectionData<T>
  list(Generator<T> gen, int quantity) {
    return new CollectionData<T>(gen, quantity);
  }
} ///:~
```

Questo codice utilizza il **Generator** per includere nel contenitore tutti gli oggetti che vi occorrono. Il contenitore risultante può poi essere passato al costruttore di qualsiasi **Collection**, e quel costruttore copierà i dati in se stesso. Anche il metodo **addAll()**, presente in ogni sottotipo di **Collection**, può essere utilizzato per popolare una **Collection** esistente.

Il metodo generico di comodo riduce la quantità di codice necessaria per utilizzare la classe. **CollectionData** è un esempio del modello *Adapter*, che adatta un **Generator** al costruttore di una **Collection**.[1]

L'esempio seguente inizializza un **LinkedHashSet**:

```
//: containers/CollectionDataTest.java
import java.util.*;
import net.mindview.util.*;

class Government implements Generator<String> {
  String[] foundation = ("strange women lying in ponds " +
    "distributing swords is no basis for a system of " +
    "government").split(" ");;
  private int index;
  public String next() { return foundation[index++]; }
}
```

1. Questa potrà non essere una definizione rigorosa dell'adattatore, così come definita in *Design Patterns* di Gamma, Helm, Johnson e Vlissides (Pearson Education Italia, 2002), ma l'autore ritiene che ne riprenda i concetti.

```
public class CollectionDataTest {
  public static void main(String[] args) {
    Set<String> set = new LinkedHashSet<String>(
      new CollectionData<String>(new Government(), 15));
    // Tramite un metodo di comodo:
    set.addAll(CollectionData.list(new Government(), 15));
    System.out.println(set);
  }
} /* Output:
[strange, women, lying, in, ponds, distributing, swords,
is, no, basis, for, a, system, of, government]
*///:~
```

Gli elementi sono nell'ordine di inserimento originale, perché un **LinkedHashSet** produce una lista collegata che conserva tale ordine.

Tutti i generatori definiti nel Capitolo 4 ora sono disponibili tramite l'adattatore **CollectionData**. L'esempio seguente si serve di due di essi:

```
//: containers/CollectionDataGeneration.java
// Utilizzo dei Generator definiti nel capitolo 4.
import java.util.*;
import net.mindview.util.*;

public class CollectionDataGeneration {
  public static void main(String[] args) {
    System.out.println(new ArrayList<String>(
      CollectionData.list( // Metodo di comodo
        new RandomGenerator.String(9), 10)));
    System.out.println(new HashSet<Integer>(
      new CollectionData<Integer>(
        new RandomGenerator.Integer(), 10)));
  }
} /* Output:
[YNzbrnyGc, FOWZnTcQr, GseGZMmJM, RoEsuEcUO, neOEdLsmw,
HLGEahKcx, rEqUCBbkI, naMesbtWH, kjUrUkZPg, wsqPzDyCy]
[573, 4779, 871, 4367, 6090, 7882, 2017, 8037, 3455, 299]
*///:~
```

La lunghezza **String** prodotta da **RandomGenerator.String** è gestita dall'argomento del costruttore.

## *Generatori di Map*

Con **Map** potete adottare lo stesso approccio, ma è richiesta una classe **Pair**, tenuto conto che per popolare una **Map** per ogni chiamata al metodo **next()** di **Generator** deve essere prodotta una coppia di oggetti, una chiave e un valore:

```
//: net/mindview/util/Pair.java
package net.mindview.util;

public class Pair<K,V> {
  public final K key;
  public final V value;
  public Pair(K k, V v) {
    key = k;
    value = v;
  }
} ///:~
```

I campi **value** e **key** sono **public** e **final**, in modo che **Pair** diventi un *oggetto DTO* (*Data Transfer Object* o *Messenger*) di sola lettura.

Ora l'adattatore **Map** può utilizzare le varie combinazioni di **Generator**, **Iterable** e valori costanti per riempire gli oggetti di inizializzazione **Map**:

```
//: net/mindview/util/MapData.java
// Una Map popolata utilizzando un oggetto generator.
package net.mindview.util;
import java.util.*;

public class MapData<K,V> extends LinkedHashMap<K,V> {
  // Un solo generatore di Pair:
  public MapData(Generator<Pair<K,V>> gen, int quantity) {
    for(int i = 0; i < quantity; i++) {
      Pair<K,V> p = gen.next();
      put(p.key, p.value);
    }
  }
```

```
// Due generatori separati:
public MapData(Generator<K> genK, Generator<V> genV,
    int quantity) {
  for(int i = 0; i < quantity; i++) {
    put(genK.next(), genV.next());
  }
}
// Un generatore di chiave e un solo valore:
public MapData(Generator<K> genK, V value, int quantity){
  for(int i = 0; i < quantity; i++) {
    put(genK.next(), value);
  }
}
// Un oggetto Iterable e un generatore di valori:
public MapData(Iterable<K> genK, Generator<V> genV) {
  for(K key : genK) {
    put(key, genV.next());
  }
}
// Un Iterable e un solo valore:
public MapData(Iterable<K> genK, V value) {
  for(K key : genK) {
    put(key, value);
  }
}
// Metodi di comodo di uso generico:
public static <K,V> MapData<K,V>
map(Generator<Pair<K,V>> gen, int quantity) {
  return new MapData<K,V>(gen, quantity);
}
public static <K,V> MapData<K,V>
map(Generator<K> genK, Generator<V> genV, int quantity) {
  return new MapData<K,V>(genK, genV, quantity);
}
public static <K,V> MapData<K,V>
map(Generator<K> genK, V value, int quantity) {
```

```
    return new MapData<K,V>(genK, value, quantity);
  }
  public static <K,V> MapData<K,V>
  map(Iterable<K> genK, Generator<V> genV) {
    return new MapData<K,V>(genK, genV);
  }
  public static <K,V> MapData<K,V>
  map(Iterable<K> genK, V value) {
    return new MapData<K,V>(genK, value);
  }
} ///:~
```

Tenete presente che avete la possibilità di scegliere se utilizzare un solo **Generator<Pair<K, V>>**, due **Generator** separati, un **Generator** e un valore costante, un **Iterable** (che include tutta la **Collection**) e un **Generator**, oppure un **Iterable** e un singolo valore.

I metodi generici di comodo riducono la quantità di codice necessaria per creare un oggetto **MapData**.

Considerate l'esempio seguente, si serve di **MapData**. Anche **Letters Generator** implementa **Iterable** producendo un **Iterator**, e può quindi essere utilizzato per testare i metodi di **MapData.map()** che funzionano con un **Iterable**:

```
//: containers/MapDataTest.java
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

class Letters implements Generator<Pair<Integer,String>>,
  Iterable<Integer> {
  private int size = 9;
  private int number = 1;
  private char letter = 'A';
  public Pair<Integer,String> next() {
    return new Pair<Integer,String>(
      number++, "" + letter++);
  }
  public Iterator<Integer> iterator() {
```

```
    return new Iterator<Integer>() {
      public Integer next() { return number++; }
      public boolean hasNext() { return number < size; }
      public void remove() {
        throw new UnsupportedOperationException();
      }
    };
  }
}

public class MapDataTest {
  public static void main(String[] args) {
    // Generatore di Pair:
    print(MapData.map(new Letters(), 11));
    // Due generatori separati:
    print(MapData.map(new CountingGenerator.Character(),
      new RandomGenerator.String(3), 8));
    // Un generatore di chiave e un solo valore:
    print(MapData.map(new CountingGenerator.Character(),
      "Value", 6));
    // Un oggetto Iterable e un generatore di valori:
    print(MapData.map(new Letters(),
      new RandomGenerator.String(3)));
    // Un Iterable e un solo valore:
    print(MapData.map(new Letters(), "Pop"));
  }
} /* Output:
{1=A, 2=B, 3=C, 4=D, 5=E, 6=F, 7=G, 8=H, 9=I, 10=J, 11=K}
{a=YNz, b=brn, c=yGc, d=FOW, e=ZnT, f=cQr, g=Gse, h=GZM}
{a=Value, b=Value, c=Value, d=Value, e=Value, f=Value}
{1=mJM, 2=RoE, 3=suE, 4=cUO, 5=neO, 6=EdL, 7=smw, 8=HLG}
{1=Pop, 2=Pop, 3=Pop, 4=Pop, 5=Pop, 6=Pop, 7=Pop, 8=Pop}
*///:~
```

Anche questo esempio si serve dei generatori che avete visto nel Capitolo 4.

Potete creare qualunque insieme di dati generato per **Map** o **Collection** servendovi di questi strumenti, e inizializzando in seguito una **Map** o una **Col-**

**lection** per mezzo del costruttore o dei metodi **Collection.addAll()** o **Map.putAll()**.

## *Utilizzo delle classi astratte*

Un metodo alternativo al problema di creare dati di prova per i contenitori consiste nel generare implementazioni personalizzate di **Map** e **Collection**.

Ogni contenitore di **java.util** ha una classe **Abstract** che fornisce un'implementazione parziale del contenitore stesso: in questo modo, per produrre il vostro contenitore non dovete fare altro che implementare i metodi che vi occorrono. Se il contenitore risultante deve essere di sola lettura, come in genere nel caso dei dati di prova, il numero di metodi da fornire è ridotto al minimo.

Benché non sia strettamente richiesto in questo caso, la soluzione illustrata di seguito fornisce l'occasione per dimostrare l'utilizzo di un nuovo design pattern: *Flyweight*. Questo modello è utile qualora la soluzione normale richieda troppi oggetti o quando la realizzazione di oggetti normali richieda troppo spazio di archiviazione. Il pattern Flyweight "esternalizza" parte dell'oggetto per evitare che esso contenga tutti i dati che gli competono, in modo che sia possibile trasferire tutti o parte di questi dati in una tabella esterna più efficiente, oppure implementare algoritmi di ottimizzazione dello spazio.

Un obiettivo importante di questo esempio è dimostrare che è relativamente semplice creare una **Map** e una **Collection** personalizzate, ereditando dalle classi di **java.util.Abstract**.

Per creare una **Map** di sola lettura, create una classe per ereditarietà da **AbstractMap** e implementate il metodo **entrySet()**. Per creare un **Set** di sola lettura, ereditate da **AbstractSet** e implementate i metodi **iterator()** e **size()**.

L'insieme di dati in questo esempio è una **Map** dei paesi del mondo e delle loro capitali.[2]

Il metodo **capitals()** produce una **Map** di nazioni e capitali, mentre il metodo **names()** elabora una **List** con i nomi delle nazioni. In entrambi i casi, potete ottenere un elenco parziale fornendo un argomento **int** che indica la quantità di dati desiderata:

```
//: net/mindview/util/Countries.java
// Mappe e liste con dati di esempio (con modello Flyweight).
package net.mindview.util;
```

2. Questi dati geografici sono stati trovati su Internet: le varie correzioni sono state segnalate dai lettori.

```
import java.util.*;
import static net.mindview.util.Print.*;

public class Countries {
  public static final String[][] DATA = {
    // Africa
    {"ALGERIA","Algiers"}, {"ANGOLA","Luanda"},
    {"BENIN","Porto-Novo"}, {"BOTSWANA","Gaberone"},
    {"BURKINA FASO","Ouagadougou"},
    {"BURUNDI","Bujumbura"},
    {"CAMEROON","Yaounde"}, {"CAPE VERDE","Praia"},
    {"CENTRAL AFRICAN REPUBLIC","Bangui"},
    {"CHAD","N'djamena"},  {"COMOROS","Moroni"},
    {"CONGO","Brazzaville"}, {"DJIBOUTI","Dijibouti"},
    {"EGYPT","Cairo"}, {"EQUATORIAL GUINEA","Malabo"},
    {"ERITREA","Asmara"}, {"ETHIOPIA","Addis Ababa"},
    {"GABON","Libreville"}, {"THE GAMBIA","Banjul"},
    {"GHANA","Accra"}, {"GUINEA","Conakry"},
    {"BISSAU","Bissau"},
    {"COTE D'IVOIRE (IVORY COAST)","Yamoussoukro"},
    {"KENYA","Nairobi"}, {"LESOTHO","Maseru"},
    {"LIBERIA","Monrovia"}, {"LIBYA","Tripoli"},
    {"MADAGASCAR","Antananarivo"}, {"MALAWI","Lilongwe"},
    {"MALI","Bamako"}, {"MAURITANIA","Nouakchott"},
    {"MAURITIUS","Port Louis"}, {"MOROCCO","Rabat"},
    {"MOZAMBIQUE","Maputo"}, {"NAMIBIA","Windhoek"},
    {"NIGER","Niamey"}, {"NIGERIA","Abuja"},
    {"RWANDA","Kigali"},
    {"SAO TOME E PRINCIPE","Sao Tome"},
    {"SENEGAL","Dakar"}, {"SEYCHELLES","Victoria"},
    {"SIERRA LEONE","Freetown"}, {"SOMALIA","Mogadishu"},
    {"SOUTH AFRICA","Pretoria/Cape Town"},
    {"SUDAN","Khartoum"},
    {"SWAZILAND","Mbabane"}, {"TANZANIA","Dodoma"},
    {"TOGO","Lome"}, {"TUNISIA","Tunis"},
    {"UGANDA","Kampala"},
```

```
{"DEMOCRATIC REPUBLIC OF THE CONGO (ZAIRE)",
 "Kinshasa"},
{"ZAMBIA","Lusaka"}, {"ZIMBABWE","Harare"},
// Asia
{"AFGHANISTAN","Kabul"}, {"BAHRAIN","Manama"},
{"BANGLADESH","Dhaka"}, {"BHUTAN","Thimphu"},
{"BRUNEI","Bandar Seri Begawan"},
{"CAMBODIA","Phnom Penh"},
{"CHINA","Beijing"}, {"CYPRUS","Nicosia"},
{"INDIA","New Delhi"}, {"INDONESIA","Jakarta"},
{"IRAN","Tehran"}, {"IRAQ","Baghdad"},
{"ISRAEL","Jerusalem"}, {"JAPAN","Tokyo"},
{"JORDAN","Amman"}, {"KUWAIT","Kuwait City"},
{"LAOS","Vientiane"}, {"LEBANON","Beirut"},
{"MALAYSIA","Kuala Lumpur"}, {"THE MALDIVES","Male"},
{"MONGOLIA","Ulan Bator"},
{"MYANMAR (BURMA)","Rangoon"},
{"NEPAL","Katmandu"}, {"NORTH KOREA","P'yongyang"},
{"OMAN","Muscat"}, {"PAKISTAN","Islamabad"},
{"PHILIPPINES","Manila"}, {"QATAR","Doha"},
{"SAUDI ARABIA","Riyadh"}, {"SINGAPORE","Singapore"},
{"SOUTH KOREA","Seoul"}, {"SRI LANKA","Colombo"},
{"SYRIA","Damascus"},
{"TAIWAN (REPUBLIC OF CHINA)","Taipei"},
{"THAILAND","Bangkok"}, {"TURKEY","Ankara"},
{"UNITED ARAB EMIRATES","Abu Dhabi"},
{"VIETNAM","Hanoi"}, {"YEMEN","Sana'a"},
// Australia e Oceania
{"AUSTRALIA","Canberra"}, {"FIJI","Suva"},
{"KIRIBATI","Bairiki"},
{"MARSHALL ISLANDS","Dalap-Uliga-Darrit"},
{"MICRONESIA","Palikir"}, {"NAURU","Yaren"},
{"NEW ZEALAND","Wellington"}, {"PALAU","Koror"},
{"PAPUA NEW GUINEA","Port Moresby"},
{"SOLOMON ISLANDS","Honaira"}, {"TONGA","Nuku'alofa"},
{"TUVALU","Fongafale"}, {"VANUATU","< Port-Vila"},
```

```
{"WESTERN SAMOA","Apia"},
// Europa orientale ed ex-URSS
{"ARMENIA","Yerevan"}, {"AZERBAIJAN","Baku"},
{"BELARUS (BYELORUSSIA)","Minsk"},
{"GEORGIA","Tbilisi"},
{"KAZAKSTAN","Almaty"}, {"KYRGYZSTAN","Alma-Ata"},
{"MOLDOVA","Chisinau"}, {"RUSSIA","Moscow"},
{"TAJIKISTAN","Dushanbe"}, {"TURKMENISTAN","Ashkabad"},
{"UKRAINE","Kyiv"}, {"UZBEKISTAN","Tashkent"},
// Europa
{"ALBANIA","Tirana"}, {"ANDORRA","Andorra la Vella"},
{"AUSTRIA","Vienna"}, {"BELGIUM","Brussels"},
{"BULGARIA","Sofia"}, {"BOSNIA","-"}, {"HERZEGOVINA","Sara
jevo"},
{"CROATIA","Zagreb"}, {"CZECH REPUBLIC","Prague"},
{"DENMARK","Copenhagen"}, {"ESTONIA","Tallinn"},
{"FINLAND","Helsinki"}, {"FRANCE","Paris"},
{"GERMANY","Berlin"}, {"GREECE","Athens"},
{"HUNGARY","Budapest"}, {"ICELAND","Reykjavik"},
{"IRELAND","Dublin"}, {"ITALY","Rome"},
{"LATVIA","Riga"}, {"LIECHTENSTEIN","Vaduz"},
{"LITHUANIA","Vilnius"}, {"LUXEMBOURG","Luxembourg"},
{"MACEDONIA","Skopje"}, {"MALTA","Valletta"},
{"MONACO","Monaco"}, {"MONTENEGRO","Podgorica"},
{"THE NETHERLANDS","Amsterdam"}, {"NORWAY","Oslo"},
{"POLAND","Warsaw"}, {"PORTUGAL","Lisbon"},
{"ROMANIA","Bucharest"}, {"SAN MARINO","San Marino"},
{"SERBIA","Belgrade"}, {"SLOVAKIA","Bratislava"},
{"SLOVENIA","Ljuijana"}, {"SPAIN","Madrid"},
{"SWEDEN","Stockholm"}, {"SWITZERLAND","Berne"},
{"UNITED KINGDOM","London"}, {"VATICAN CITY","---"},
// America settentrionale e centrale
{"ANTIGUA AND BARBUDA","Saint John's"},
{"BAHAMAS","Nassau"},
{"BARBADOS","Bridgetown"}, {"BELIZE","Belmopan"},
{"CANADA","Ottawa"}, {"COSTA RICA","San Jose"},
{"CUBA","Havana"}, {"DOMINICA","Roseau"},
```

```
    {"DOMINICAN REPUBLIC","Santo Domingo"},
    {"EL SALVADOR","San Salvador"},
    {"GRENADA","Saint George's"},
    {"GUATEMALA","Guatemala City"},
    {"HAITI","Port-au-Prince"},
    {"HONDURAS","Tegucigalpa"}, {"JAMAICA","Kingston"},
    {"MEXICO","Mexico City"}, {"NICARAGUA","Managua"},
    {"PANAMA","Panama City"}, {"ST. KITTS","-"},
    {"NEVIS","Basseterre"}, {"ST. LUCIA","Castries"},
    {"ST. VINCENT AND THE GRENADINES","Kingstown"},
    {"UNITED STATES OF AMERICA","Washington, D.C."},
    // America meridionale
    {"ARGENTINA","Buenos Aires"},
    {"BOLIVIA","Sucre (legale)/La Paz(amministrativa)"},
    {"BRAZIL","Brasilia"}, {"CHILE","Santiago"},
    {"COLOMBIA","Bogota"}, {"ECUADOR","Quito"},
    {"GUYANA","Georgetown"}, {"PARAGUAY","Asuncion"},
    {"PERU","Lima"}, {"SURINAME","Paramaribo"},
    {"TRINIDAD AND TOBAGO","Port of Spain"},
    {"URUGUAY","Montevideo"}, {"VENEZUELA","Caracas"},
  };
  // Utilizzo di AbstractMap mediante implementazione di
  // entrySet()
  private static class FlyweightMap
  extends AbstractMap<String,String> {
    private static class Entry
    implements Map.Entry<String,String> {
      int index;
      Entry(int index) { this.index = index; }
      public boolean equals(Object o) {
        return DATA[index][0].equals(o);
      }
      public String getKey() { return DATA[index][0]; }
      public String getValue() { return DATA[index][1]; }
      public String setValue(String value) {
        throw new UnsupportedOperationException();
      }
```

```
    public int hashCode() {
      return DATA[index][0].hashCode();
    }
  }
  // Utilizzo di AbstractSet mediante implementazione di
  // size() e iterator()
  static class EntrySet
  extends AbstractSet<Map.Entry<String,String>> {
    private int size;
    EntrySet(int size) {
      if(size < 0)
        this.size = 0;
      // Non puo' essere maggiore dell'array:
      else if(size > DATA.length)
        this.size = DATA.length;
      else
        this.size = size;
    }
    public int size() { return size; }
    private class Iter
    implements Iterator<Map.Entry<String,String>> {
      // Soltanto un oggetto Entry per ogni Iterator:
      private Entry entry = new Entry(-1);
      public boolean hasNext() {
        return entry.index < size - 1;
      }
      public Map.Entry<String,String> next() {
        entry.index++;
        return entry;
      }
      public void remove() {
        throw new UnsupportedOperationException();
      }
    }
    public
    Iterator<Map.Entry<String,String>> iterator() {
      return new Iter();
```

```
    }
  }
  private static Set<Map.Entry<String,String>> entries =
    new EntrySet(DATA.length);
  public Set<Map.Entry<String,String>> entrySet() {
    return entries;
  }
}
// Crea una mappa partial di N paesi (N = size):
static Map<String,String> select(final int size) {
  return new FlyweightMap() {
    public Set<Map.Entry<String,String>> entrySet() {
      return new EntrySet(size);
    }
  };
}
static Map<String,String> map = new FlyweightMap();
public static Map<String,String> capitals() {
  return map; // L'intera mappa
}
public static Map<String,String> capitals(int size) {
  return select(size); // Una mappa parziale
}
static List<String> names =
  new ArrayList<String>(map.keySet());
// Tutti i nomi:
public static List<String> names() { return names; }
// Un elenco parziale:
public static List<String> names(int size) {
  return new ArrayList<String>(select(size).keySet());
}
public static void main(String[] args) {
  print(capitals(10));
  print(names(10));
  print(new HashMap<String,String>(capitals(3)));
  print(new LinkedHashMap<String,String>(capitals(3)));
  print(new TreeMap<String,String>(capitals(3)));
```

```
    print(new Hashtable<String,String>(capitals(3)));
    print(new HashSet<String>(names(6)));
    print(new LinkedHashSet<String>(names(6)));
    print(new TreeSet<String>(names(6)));
    print(new ArrayList<String>(names(6)));
    print(new LinkedList<String>(names(6)));
    print(capitals().get("BRAZIL"));
  }
} /* Output:
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-
Novo, BOTSWANA=Gaberone, BURKINA FASO=Ouagadougou,
BURUNDI=Bujumbura, CAMEROON=Yaounde, CAPE VERDE=Praia, CENTRAL
AFRICAN REPUBLIC=Bangui, CHAD=N'djamena}
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI,
CAMEROON, CAPE VERDE, CENTRAL AFRICAN REPUBLIC, CHAD]
{BENIN=Porto-Novo, ANGOLA=Luanda, ALGERIA=Algiers}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
[BURKINA FASO, BOTSWANA, BENIN, ANGOLA, ALGERIA, BURUNDI]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI]
Brasilia
*///:~
```

L'array bidimensionale di **String DATA** è **public**, per essere utilizzabile altrove. **FlyweightMap** deve implementare il metodo **entrySet()**, che richiede un'implementazione **Set** e una classe **Map.Entry**, entrambe personalizzate. Ecco come funziona la prima parte del modello Flyweight: ogni oggetto **Map.Entry** memorizza semplicemente l'indice, invece degli effettivi chiave e valore; quando chiamate **getKey()** o **getValue()**, il pattern si serve dell'indice per restituire l'elemento **DATA** appropriato. La classe **EntrySet** garantisce che le sue dimensioni (**size**) non eccedano quelle dei dati (**DATA**).

L'altra parte del pattern Flyweight è implementata in **EntrySet.Iterator**: invece di creare un oggetto **Map.Entry** per ogni coppia di dati in **DATA**, esiste un solo oggetto **Map.Entry** *per ogni iteratore*. L'oggetto **Entry** serve come "fine-

stra" verso i dati, contenendo unicamente un indice (**index**) nell'array statico di stringhe. Ogni volta che chiamerete **next()** sull'iteratore, l'indice (**index**) di **Entry** verrà incrementato per indicare la successiva coppia di elementi, ed è a quel punto che l'oggetto **Entry** univoco di **Iterator** sarà restituito da **next()**.[3]

Il metodo **select()** produce una **FlyweightMap** contenente un **EntrySet** del formato voluto, che viene poi utilizzato nei metodi sovraccaricati **capitals()** e **names()** di **main()**.

Potete adottare lo stesso *modus operandi* per creare contenitori inizializzati personalizzati che abbiano un dataset di dimensioni arbitrarie. Questa classe è una **List** di dimensioni qualsiasi, preinizializzata con dati di tipo **Integer**:

```
//: net/mindview/util/CountingIntegerList.java
// List di dimensioni arbitrarie, contenente dati campione.
package net.mindview.util;
import java.util.*;

public class CountingIntegerList
extends AbstractList<Integer> {
  private int size;
  public CountingIntegerList(int size) {
    this.size = size < 0 ? 0 : size;
  }
  public Integer get(int index) {
    return Integer.valueOf(index);
  }
  public int size() { return size; }
  public static void main(String[] args) {
    System.out.println(new CountingIntegerList(30));
  }
} /* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
*///:~
```

3. Per tutte le **Map** in **java.util**, le copie dei dati vengono eseguite servendosi di metodi standard **getKey()** e **getValue()** per le **Map**; in effetti, se una **Map** personalizzata dovesse semplicemente copiare l'intero contenuto di **Map.Entry**, si potrebbero avere problemi in caso di grandi quantità di dati.

Per creare una **List** di sola lettura da una **AbstractList**, dovete implementare **get()** e **size()**. Anche in questo caso viene utilizzata una soluzione Flyweight: è **get()** che si occupa di fornire il valore richiesto, in modo da non dovere effettivamente popolare la **List**. Considerate questa **Map**, anch'essa di dimensioni arbitrarie, che contiene valori univoci **Integer** e **String** preinizializzati:

```
//: net/mindview/util/CountingMapData.java
// Map di dimensioni illimitate, contenente dati campione.
package net.mindview.util;
import java.util.*;

public class CountingMapData
extends AbstractMap<Integer,String> {
  private int size;
  private static String[] chars =
    "A B C D E F G H I J K L M N O P Q R S T U V W X Y Z"
    .split(" ");
  public CountingMapData(int size) {
    if(size < 0) this.size = 0;
    this.size = size;
  }
  private static class Entry
  implements Map.Entry<Integer,String> {
    int index;
    Entry(int index) { this.index = index; }
    public boolean equals(Object o) {
      return Integer.valueOf(index).equals(o);
    }
    public Integer getKey() { return index; }
    public String getValue() {
      return
        chars[index % chars.length] +
        Integer.toString(index / chars.length);
    }
    public String setValue(String value) {
      throw new UnsupportedOperationException();
    }
```

```
      public int hashCode() {
        return Integer.valueOf(index).hashCode();
      }
    }
    public Set<Map.Entry<Integer,String>> entrySet() {
      // LinkedHashSet mantiene l'ordine d'inizializzazione:
      Set<Map.Entry<Integer,String>> entries =
        new LinkedHashSet<Map.Entry<Integer,String>>();
      for(int i = 0; i < size; i++)
        entries.add(new Entry(i));
      return entries;
    }
    public static void main(String[] args) {
      System.out.println(new CountingMapData(60));
    }
  } /* Output:
  {0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0, 9=J0,
  10=K0, 11=L0, 12=M0, 13=N0, 14=O0, 15=P0, 16=Q0, 17=R0, 18=S0,
  19=T0, 20=U0, 21=V0, 22=W0, 23=X0, 24=Y0, 25=Z0, 26=A1, 27=B1,
  28=C1, 29=D1, 30=E1, 31=F1, 32=G1, 33=H1, 34=I1, 35=J1, 36=K1,
  37=L1, 38=M1, 39=N1, 40=O1, 41=P1, 42=Q1, 43=R1, 44=S1, 45=T1,
  46=U1, 47=V1, 48=W1, 49=X1, 50=Y1, 51=Z1, 52=A2, 53=B2, 54=C2,
  55=D2, 56=E2, 57=F2, 58=G2, 59=H2}
  *///:~
```

Tenete presente che in questo codice il pattern Flyweight non è implementato in modo completo, poiché invece di creare una classe **Set** personalizzata si utilizza un **LinkedHashSet**.

**Esercizio 1** (1) Create una **List**, provando sia **ArrayList** sia **LinkedList**, e popolatela utilizzando **Countries**. Ordinate la lista e visualizzatela, quindi applicate ripetutamente **Collections.shuffle()** alla lista, visualizzandola ogni volta in modo da vedere come il metodo **shuffle()** presenti i dati dell'elenco in modo casuale.

**Esercizio 2** (2) Generate una **Map** e un **Set** che contenga tutte le nazioni il cui nome inizia con la lettera 'A'.

**Esercizio 3** (1) Servendovi di **Countries**, popolate più volte un **Set** con gli stessi dati e verificate che nel **Set** esista soltanto un'istanza della stessa nazione. Provate questo esercizio con **HashSet**, **LinkedHashSet** e **TreeSet**.

**Esercizio 4** (2) Dimostrate che è possibile creare un inizializzatore di **Collection** che apra un file e lo suddivida in parole servendosi della classe **TextFile**, e utilizzate le singole parole così ottenute per popolare la **Collection** risultante.

**Esercizio 5** (3) Modificate **CountingMapData.java** per implementare il modello Flyweight completo, aggiungendo una classe **EntrySet** personalizzata, analoga a quella in **Countries.java**.

## Funzionalità di Collection

La tabella seguente elenca tutti i metodi ammessi per una **Collection**, eccetto quelli ereditati automaticamente da **Object**: di conseguenza queste operazioni sono permesse anche in un **Set** e in una **List**, che pure offre funzionalità aggiuntive. La classe **Map**, che non è ereditata da **Collection**, verrà trattata a parte.

| | |
|---|---|
| **boolean add(E)** | Assicura che il contenitore registri il valore passato come argomento: restituisce **true** se la collezione risulterà modificata a seguito di questa chiamata. Restituisce **false** se l'argomento non può essere aggiunto. Questo è un metodo "facoltativo", descritto nel paragrafo successivo. |
| **boolean addAll (Collection<extends E>)** | Aggiunge tutti gli elementi passati come argomento: restituisce **true** se viene inserito un elemento, **false** in caso contrario. Il metodo è "facoltativo". |
| **void clear()** | Elimina tutti gli elementi nel contenitore. Il metodo è "facoltativo". |
| **boolean contains (Object)** | Restituisce **true** se il contenitore contiene l'argomento specificato; **false** in caso contrario. |
| **boolean containsAll (Collection<?>)** | Restituisce **true** se il contenitore contiene tutti gli elementi passati come argomento. |
| **boolean isEmpty()** | Restituisce **true** se il contenitore è vuoto. |
| **Iterator<E> iterator()** | Restituisce un **Iterator** che può essere usato per spostarsi all'interno degli elementi nel contenitore. |
| **boolean remove(Object)** | Se l'elemento passato come argomento è presente nel contenitore, elimina un'istanza dell'elemento; restituisce **true** in caso di eliminazione. Il metodo è "facoltativo". |

| | |
|---|---|
| **boolean removeAll (Collection<?>)** | Elimina tutti gli elementi passati come argomento, e restituisce **true** in caso di eliminazione. Il metodo è "facoltativo". |
| **boolean retainAll (Collection<?>)** | Mantiene soltanto gli elementi che sono passati come argomento: un'intersezione, secondo la teoria degli insiemi; restituisce **true** in caso di modifica. Il metodo è "facoltativo". |
| **int size()** | Restituisce il numero di elementi presenti nel contenitore. |
| **Object[] toArray()** | Restituisce un array che contiene tutti gli elementi presenti nel contenitore. |
| **<T> T[] toArray(T[] a)** | Restituisce un array che contiene tutti gli elementi presenti nel contenitore. Il tipo a runtime del risultato è quello dell'argomento **a**, anziché il normale **Object**. |

Notate che non è disponibile un metodo **get()** per l'accesso casuale alla selezione degli elementi, in quanto **Collection** include anche **Set**, che gestisce internamente l'ordinamento: di conseguenza una ricerca ad accesso casuale sarebbe priva di senso. Quindi, per esaminare gli elementi di una **Collection** dovete ricorrere a un iteratore.

Il seguente esempio mostra tutti i metodi all'opera; tenete presente che sebbene questi metodi funzionino con qualunque oggetto implementi **Collection**, come "minimo comune denominatore" viene utilizzato un **ArrayList**:

```
//: containers/CollectionMethods.java
// Operazioni ammesse in tutte le Collection.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class CollectionMethods {
  public static void main(String[] args) {
    Collection<String> c = new ArrayList<String>();
    c.addAll(Countries.names(6));
    c.add("ten");
    c.add("eleven");
    print(c);
    // Crea un array dalla List:
```

```
Object[] array = c.toArray();
// Crea un array di String dalla List:
String[] str = c.toArray(new String[0]);
// Recupera gli elementi max e min; questa operazione assume
// significati diversi a seconda dell'implementazione
// dell'interfaccia Comparable:
print("Collections.max(c) = " + Collections.max(c));
print("Collections.min(c) = " + Collections.min(c));
// Aggiunge una Collection a un'altra Collection
Collection<String> c2 = new ArrayList<String>();
c2.addAll(Countries.names(6));
c.addAll(c2);
print(c);
c.remove(Countries.DATA[0][0]);
print(c);
c.remove(Countries.DATA[1][0]);
print(c);
// Elimina tutti i componenti presenti nella collection
// passata come argomento:
c.removeAll(c2);
print(c);
c.addAll(c2);
print(c);
// Questo elemento e' presente nella Collection corrente?
String val = Countries.DATA[3][0];
print("c.contains(" + val  + ") = " + c.contains(val));
// Questa Collection e' presente nella Collection corrente?
print("c.containsAll(c2) = " + c.containsAll(c2));
Collection<String> c3 =
  ((List<String>)c).subList(3, 5);
// Conserva tutti gli elementi presenti in c2 e c3
// (intersezione di insiemi):
c2.retainAll(c3);
print(c2);
// Elimina tutti gli elementi di c2
// che sono presenti anche in c3:
```

```
    c2.removeAll(c3);
    print("c2.isEmpty() = " +  c2.isEmpty());
    c = new ArrayList<String>();
    c.addAll(Countries.names(6));
    print(c);
    c.clear(); // Elimina tutti gli elementi
    print("after c.clear():" + c);
  }
} /* Output:
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO,
ten, eleven]
Collections.max(c) = ten
Collections.min(c) = ALGERIA
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO,
ten, eleven, ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA,
BURKINA FASO]
[ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO, ten, eleven,
ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
[BENIN, BOTSWANA, BULGARIA, BURKINA FASO, ten, eleven,
ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
[ten, eleven]
[ten, eleven, ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA,
BURKINA FASO]
c.contains(BOTSWANA) = true
c.containsAll(c2) = true
[ANGOLA, BENIN]
c2.isEmpty() = true
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
after c.clear():[]
*///:~
```

Il codice crea alcuni **ArrayList** contenenti diversi dataset e ne esegue l'upcast a oggetti **Collection**: è quindi chiaro che viene utilizzata unicamente l'interfaccia **Collection**. Il metodo **main()** esegue alcune semplici operazioni per illustrare l'utilizzo di tutti i metodi di **Collection**.

Le descrizioni delle classi *legacy* **Vector**, **Stack** e **Hashtable** figurano alla fine del capitolo: benché non dobbiate servirvene, potreste comunque trovarle nel codice di programmi esistenti.

## Operazioni facoltative

I metodi che eseguono le varie operazioni di aggiunta e rimozione sono *facoltativi* nell'interfaccia **Collection**: questo significa che la classe di implementazione non deve necessariamente fornire le definizioni di funzionamento per questi metodi.

Questo è un modo piuttosto insolito di definire un'interfaccia. Come sapete, nella programmazione OOP un'interfaccia è una sorta di contratto con il quale garantite di inviare messaggi a un determinato oggetto, a prescindere dalla tecnica utilizzata nell'implementazione dell'interfaccia.[4]

Tuttavia, un funzionamento "facoltativo" viola questo principio fondamentale, indicando che la chiamata di alcuni metodi *non* produrrà un comportamento coerente, ma solleverà alcune eccezioni! È come se la sicurezza del tipo in fase di compilazione non fosse tenuta in alcun conto.

In realtà, come si suol dire, non tutto il male viene per nuocere: se un'operazione è facoltativa, il compilatore continuerà a impedire di chiamare metodi esterni all'interfaccia. Si tratta di un comportamento diverso da quello dei linguaggi dinamici, nei quali è possibile chiamare qualunque metodo di qualsiasi oggetto, scoprendo soltanto al momento dell'esecuzione se una determinata chiamata funzionerà.[5]

Tenete presente che i metodi di lettura di **Collection** non sono facoltativi; sono metodi di lettura la maggior parte di quelli che accettano come argomento una **Collection**.

Perché mai dovreste definire alcuni metodi come "facoltativi"? Perché in questo modo impedite la potenziale moltiplicazione di interfacce in fase di progettazione.

Altre progettazioni di librerie di contenitori sembrano risolversi in una confusione di interfacce, ciascuna delle quali descrive variazioni su un tema principale. Non è neppure possibile intercettare tutti i casi speciali nelle interfacce, perché qualcuno può sempre inventarne una nuova. L'approccio dell'"operazione non supportata" realizza un obiettivo importante della libreria dei contenitori Java: grazie a questo approccio, è semplice imparare e utilizzare i contenitori.

Le operazioni non supportate sono un caso speciale che può essere ritardato fino al momento opportuno. Affinché questa tecnica funzioni, tuttavia, occorre attenersi alle regole elencate di seguito.

---

4. In questo contesto, il termine "interfaccia" definisce sia la parola chiave **interface** formale sia il significato più generale di "metodi supportati da qualsiasi classe o sottoclasse".
5. Per quanto questa affermazione possa sembrare strana, avete visto (soprattutto nel Capitolo 2) che questo genere di comportamento dinamico può rivelarsi particolarmente potente.

1. L'eccezione **UnsupportedOperationException** deve essere un evento sporadico: in pratica, per la maggior parte delle classi tutte le operazioni dovrebbero funzionare perfettamente e soltanto in casi speciali un'operazione potrebbe non essere supportata. Questo è vero per la libreria dei contenitori Java, poiché le classi che utilizzerete con maggiore frequenza, vale a dire **ArrayList**, **LinkedList**, **HashSet** e **HashMap** nonché altre implementazioni concrete, supportano tutte le operazioni. La progettazione deve potervi lasciare una "scappatoia", nel caso vogliate creare una nuova **Collection** e inserirla nella libreria attuale senza dovere obbligatoriamente fornire delle definizioni significative per tutti i metodi nell'interfaccia **Collection**.
2. Quando un'operazione *non è* supportata, dovrebbe essere auspicabile che una **UnsupportedOperationException** appaia in fase di implementazione, anziché dopo aver consegnato il software al cliente; dopotutto, l'eccezione indica un errore di programmazione derivante dall'utilizzo errato di un'implementazione.

Vale la pena notare che le operazioni non supportate sono rilevabili soltanto in fase di esecuzione, e per questo rappresentano un controllo di tipo dinamico. Se la vostra esperienza si basa prevalentemente su linguaggi di tipo statico come C++, Java potrebbe apparirvi come un linguaggio fra i tanti.

Certamente Java *ha* un controllo statico, ma anche alcune doti significative caratteristiche del controllo di tipo dinamico, e questo dualismo impedisce di collocarlo definitivamente in una categoria o nell'altra. Una volta che inizierete a notare questo dualismo, individuerete altri esempi di controllo di tipo dinamico in Java.

## Operazioni non supportate

Una fonte comune di operazioni non supportate è costituita da un contenitore basato su una struttura dati a dimensione fissa: questo contenitore si ottiene utilizzando il metodo **Arrays.asList()** per trasformare un array in una **List**.

Potete anche *decidere* di fare in modo che qualsiasi contenitore, inclusa una **Map**, sollevi eccezioni di tipo **UnsupportedOperationException**, ricorrendo ai metodi "non modificabili" della classe **Collections**. Questo esempio illustra entrambe le possibilità:

```
//: containers/Unsupported.java
// Operazioni non supportate nei contenitori Java.
import java.util.*;
```

```
public class Unsupported {
  static void test(String msg, List<String> list) {
    System.out.println("--- " + msg + " ---");
    Collection<String> c = list;
    Collection<String> subList = list.subList(1,8);
    // Copia della sottolista:
    Collection<String> c2 = new ArrayList<String>(subList);
    try { c.retainAll(c2); } catch(Exception e) {
      System.out.println("retainAll(): " + e);
    }
    try { c.removeAll(c2); } catch(Exception e) {
      System.out.println("removeAll(): " + e);
    }
    try { c.clear(); } catch(Exception e) {
      System.out.println("clear(): " + e);
    }
    try { c.add("X"); } catch(Exception e) {
      System.out.println("add(): " + e);
    }
    try { c.addAll(c2); } catch(Exception e) {
      System.out.println("addAll(): " + e);
    }
    try { c.remove("C"); } catch(Exception e) {
      System.out.println("remove(): " + e);
    }
    // Il metodo List.set() modifica il valore ma non cambia
    // le dimensioni della struttura dati:
    try {
      list.set(0, "X");
    } catch(Exception e) {
      System.out.println("List.set(): " + e);
    }
  }
  public static void main(String[] args) {
    List<String> list =
      Arrays.asList("A B C D E F G H I J K L".split(" "));
```

```
    test("Modifiable Copy", new ArrayList<String>(list));
    test("Arrays.asList()", list);
    test("unmodifiableList()",
      Collections.unmodifiableList(
        new ArrayList<String>(list)));
  }
} /* Output:
--- Modifiable Copy ---
--- Arrays.asList() ---
retainAll(): java.lang.UnsupportedOperationException
removeAll(): java.lang.UnsupportedOperationException
clear(): java.lang.UnsupportedOperationException
add(): java.lang.UnsupportedOperationException
addAll(): java.lang.UnsupportedOperationException
remove(): java.lang.UnsupportedOperationException
--- unmodifiableList() ---
retainAll(): java.lang.UnsupportedOperationException
removeAll(): java.lang.UnsupportedOperationException
clear(): java.lang.UnsupportedOperationException
add(): java.lang.UnsupportedOperationException
addAll(): java.lang.UnsupportedOperationException
remove(): java.lang.UnsupportedOperationException
List.set(): java.lang.UnsupportedOperationException
*///:~
```

Dal momento che **Arrays.asList()** produce una **List** supportata da un array a dimensione fissa, ha senso che le uniche operazioni supportate siano quelle che non modificano le *dimensioni* dell'array. Qualsiasi metodo che provocasse un cambiamento nelle dimensioni della struttura dati sottostante produrrebbe una **UnsupportedOperationException** indicante una chiamata a un metodo non supportato, ovvero un errore di programmazione.

Ricordate che potete sempre passare il risultato di **Arrays.asList()** come argomento al costruttore di qualunque **Collection**, oppure servirvi del metodo **addAll()** o del metodo **static Collections.addAll()** per creare un normale contenitore che consenta l'utilizzo di tutti i metodi: questo approccio è illustrato nella prima chiamata a **test()** di **main()**, che produce una nuova struttura dati sottostante, con caratteristiche di ridimensionabilità.

I metodi "non modificabili" nella classe **Collections** inglobano il contenitore in un proxy che produrrà un'eccezione **UnsupportedOperationException**, se eseguirete qualsiasi operazione che modifichi il contenitore. L'obiettivo dell'utilizzo di questi metodi è produrre un oggetto contenitore "costante". L'elenco dei metodi "non modificabili" di **Collections** è fornito nel prosieguo del capitolo.

L'ultimo blocco **try** in **test()** esamina il metodo **set()** che appartiene a **List**. Si tratta di una tecnica interessante, poiché mostra quanto sia pratica la granularità dell'"operazione non supportata": l'"interfaccia" risultante può variare anche di un solo metodo tra l'oggetto restituito da **Arrays.asList()** e quello restituito da **Collections.unmodifiableList()**. Infatti, il metodo **Arrays.asList()** restituisce una **List** di dimensioni fisse, ma modificabile, mentre **Collections.unmodifiableList()** produce una lista che non può essere cambiata in alcun modo. Come potete vedere dall'output, in effetti è del tutto lecito *modificare* gli elementi nella **List** restituita da **Arrays.asList()**, perché questo non viola le caratteristiche di "immutabilità dimensionale" di quella **List**. Chiaramente, però, i risultati di **unmodifiableList()** non devono essere modificabili in alcun modo. Qualora si utilizzino le interfacce, per ottenere questi risultati occorrerebbero due interfacce aggiuntive, una con un metodo operativo **set()** e una senza. Queste interfacce supplementari sarebbero richieste per i diversi sottotipi non modificabili di **Collection**.

La documentazione fornita a corredo di un metodo che accetta come argomento un contenitore dovrebbe specificare quale dei metodi facoltativi dovrà essere implementato.

> **Esercizio 6** (2) Notate che **List** offre altre operazioni "opzionali" che non sono incluse in **Collection**. Realizzate una versione di **Unsupported.java** che testi queste operazioni supplementari.

## Funzionalità di List

Come avete visto, la classe **List** di base è abbastanza semplice da utilizzare: generalmente basta chiamare **add()** per inserire alcuni oggetti, **get()** per recuperarne uno alla volta e **iterator()** per ottenere un **Iterator** dalla sequenza.

Ciascuno dei metodi illustrati nel codice seguente si riferisce a un diverso gruppo di attività.

1. Operazioni che qualsiasi **List** può svolgere (**basicTest()**).
2. Confronto tra metodi di spostamento e di modifica forniti da un **Iterator** (raffronto tra **iterMotion()** e **iterManipulation()**).
3. Visualizzazione degli effetti di manipolazione di una **List** (**testVisual()**).

**4.** Operazioni disponibili soltanto alle **LinkedList.**

```
//: containers/Lists.java
// Operazioni ammesse nelle List.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class Lists {
  private static boolean b;
  private static String s;
  private static int i;
  private static Iterator<String> it;
  private static ListIterator<String> lit;
  public static void basicTest(List<String> a) {
    a.add(1, "x"); // Aggiunge un elemento in posizione 1
    a.add("x"); // Accoda un elemento
    // Aggiunta di una Collection:
    a.addAll(Countries.names(25));
    // Aggiunta di una Collection che inizia nella posizione 3:
    a.addAll(3, Countries.names(25));
    b = a.contains("1"); // Esiste, qui?
    // Esiste tutta la Collection, qui?
    b = a.containsAll(Countries.names(25));
    // Le List permettono l'accesso casuale, che e' vantaggioso
    // per gli ArrayList, e oneroso per le LinkedList:
    s = a.get(1); // Ottiene l'oggetto (tipizzato) presente
                  // in posizione 1
    i = a.indexOf("1"); // Fornisce l'indice di un elemento
    b = a.isEmpty(); // Contiene qualche elemento?
    it = a.iterator(); // Iterator normale
    lit = a.listIterator(); // ListIterator
    lit = a.listIterator(3); // Inizia nella posizione 3
    i = a.lastIndexOf("1"); // Ultima ricorrenza dell'elemento
                            // specificato
    a.remove(1); // Elimina la posizione 1
```

```
    a.remove("3"); // Elimina l'oggetto corrente
    a.set(1, "y"); // Imposta la posizione 1 a "y"
    // Conserva qualsiasi cosa sia presente nell'argomento,
    // rimuovendo il resto
    // (intersezione di due insiemi):
    a.retainAll(Countries.names(25));
    // Elimina tutto cio' che e' presente nell'argomento:
    a.removeAll(Countries.names(25));
    i = a.size(); // Che dimensioni ha?
    a.clear(); // Elimina tutti gli elementi
  }
  public static void iterMotion(List<String> a) {
    ListIterator<String> it = a.listIterator();
    b = it.hasNext();
    b = it.hasPrevious();
    s = it.next();
    i = it.nextIndex();
    s = it.previous();
    i = it.previousIndex();
  }
  public static void iterManipulation(List<String> a) {
    ListIterator<String> it = a.listIterator();
    it.add("47");
    // Bisogna spostarsi su un elemento dopo add():
    it.next();
    // Elimina l'elemento successivo a quello appena prodotto:
    it.remove();
    // Bisogna spostarsi su un elemento dopo remove():
    it.next();
    // Modifica l'elemento successivo a quello cancellato:
    it.set("47");
  }
  public static void testVisual(List<String> a) {
    print(a);
    List<String> b = Countries.names(25);
    print("b = " + b);
```

```
    a.addAll(b);
    a.addAll(b);
    print(a);
    // Inserisce, elimina e sostituisce gli elementi
    // utilizzando ListIterator:
    ListIterator<String> x = a.listIterator(a.size()/2);
    x.add("one");
    print(a);
    print(x.next());
    x.remove();
    print(x.next());
    x.set("47");
    print(a);
    // Scorre la lista in senso inverso:
    x = a.listIterator(a.size());
    while(x.hasPrevious())
      printnb(x.previous() + " ");
    print();
    print("testVisual finished");
  }
  // Esistono alcune operazioni possibili soltanto con le
  // LinkedList:
  public static void testLinkedList() {
    LinkedList<String> ll = new LinkedList<String>();
    ll.addAll(Countries.names(25));
    print(ll);
    // Simula uno stack, in "push":
    ll.addFirst("one");
    ll.addFirst("two");
    print(ll);
    // Equivale al "peek" in cima allo stack:
    print(ll.getFirst());
    // Equivale al "pop" di uno stack:
    print(ll.removeFirst());
    print(ll.removeFirst());
    // Simula una coda, estraendo gli elementi
    // a partire dal fondo:
```

```
    print(ll.removeLast());
    print(ll);
  }
  public static void main(String[] args) {
    // Crea e popola ogni volta una nuova lista:
    basicTest(
      new LinkedList<String>(Countries.names(25)));
    basicTest(
      new ArrayList<String>(Countries.names(25)));
    iterMotion(
      new LinkedList<String>(Countries.names(25)));
    iterMotion(
      new ArrayList<String>(Countries.names(25)));
    iterManipulation(
      new LinkedList<String>(Countries.names(25)));
    iterManipulation(
      new ArrayList<String>(Countries.names(25)));
    testVisual(
      new LinkedList<String>(Countries.names(25)));
    testLinkedList();
  }
} /* (Da eseguire per visualizzare l'output) *///:~
```

In **basicTest()** e **iterMotion()** le chiamate vengono eseguite per mostrare la sintassi corretta e, sebbene venga intercettato, il valore di ritorno non è utilizzato; in alcuni casi, peraltro, il valore di ritorno non viene neppure intercettato. Tenete presente che prima di utilizzare questi metodi dovreste sempre avere l'accortezza di consultare la documentazione JDK.

**Esercizio 7** (4) Create un **ArrayList** e una **LinkedList**, e popolatele usando per ognuna il generatore **Countries.names()**. Visualizzate le due liste mediante un normale **Iterator**, poi inserite una lista nell'altra ricorrendo a un'interfaccia **ListIterator**, senza sovrascrivere gli elementi esistenti. Eseguite infine l'inserimento a partire dalla fine della prima lista e procedete in senso inverso.

**Esercizio 8** (7) Create una classe lista generica "a collegamento singolo" (*singly linked list*) chiamata **SList**: per semplicità, questa lista *non* implementa l'interfaccia **List**. Ogni oggetto **Link** nella lista dovrà contenere un *riferimento* al successivo elemento, ma non a quello pre-

cedente: tenete presente che, al contrario, una **LinkedList** è una lista "a collegamento doppio" (*doubly linked list*), perché mantiene i collegamenti in entrambe le direzioni. Create il vostro **SListIterator** personalizzato che, sempre per semplicità, non implementi **ListIterator**. Oltre a **toString()**, l'unico metodo in **SList** dovrebbe essere **iterator()**, che produce un **SListIterator**. L'inserimento e la rimozione degli elementi da una **SList** dovrà essere possibile solo mediante **SListIterator**. Scrivete il codice che dimostri il corretto funzionamento di **SList**.

## Set e ordine di archiviazione

Pur essendo una valida introduzione alle operazioni eseguibili con i **Set** di base, gli esempi di **Set** presentati nel Volume 1, Capitolo 11 si servono di tipi Java predefiniti, quali **Integer** e **String**, che sono stati concepiti per essere utilizzati all'interno dei contenitori.

Nel creare i vostri tipi personalizzati dovete essere consapevoli che un **Set** necessita di un metodo per conservare l'ordine di archiviazione, e che tale metodo è variabile in funzione delle diverse implementazioni di **Set**. Quindi, implementazioni differenti di **Set** non solo avranno comportamenti differenti, ma anche esigenze diverse per quanto riguarda il tipo di oggetto che può essere inserito in un determinato **Set**.

| | |
|---|---|
| **Set** (interfaccia) | Ogni elemento aggiunto al **Set** deve essere univoco, perché non è ammessa l'aggiunta di elementi duplicati. Gli elementi aggiunti al **Set** devono implementare **equals()** per stabilire la loro univocità.<br>L'oggetto **Set** ha esattamente la stessa interfaccia di **Collection**. Non è assicurato che l'interfaccia di **Set** mantenga gli elementi in alcun ordine particolare. |
| **HashSet*** | Per i **Set** nei quali sia importante garantire tempi di consultazione brevi. Gli elementi devono definire anche il metodo **hashCode()**. |
| **TreeSet** | Un **Set** ordinato e supportato da un'alberatura, che vi consente di estrarre una sequenza di elementi ordinata. Gli elementi devono implementare anche l'interfaccia **Comparable**. |
| **LinkedHashSet** | Consente la rapida ricerca di un **HashSet**, ma internamente mantiene l'ordine d'inserimento degli elementi grazie a una lista collegata: di conseguenza, quando iterate il **Set** i risultati compaiono nell'ordine di inserimento. Gli elementi devono definire anche il metodo **hashCode()**. |

L'asterisco in corrispondenza di **HashSet** indica che, in assenza di altri vincoli, questa dovrebbe essere la vostra scelta predefinita, essendo ottimizzata in termini di velocità.

La definizione del metodo **hashCode()** verrà descritta nel prosieguo del capitolo. È indispensabile creare un metodo **equals()** sia per i metodi di archiviazione basati su hash (**HashSet** e **LinkedHashSet**) sia per quello supportato da un'alberatura (**TreeSet**); **hashCode()** è richiesto invece soltanto se la classe sarà collocata in un **HashSet** (come è probabile, dal momento che questa dovrebbe essere la vostra implementazione **Set** predefinita) o in un **LinkedHashSet**. In ogni caso, per rispettare uno stile di programmazione corretto, dovreste sempre sovrascrivere **hashCode()** quando sovrascrivete **equals()**.

Questo esempio illustra i metodi che devono essere definiti per utilizzare in modo efficace un tipo con un'implementazione particolare di **Set**:

```
//: containers/TypesForSets.java
// Metodi richiesti per l'inserimento di un tipo
// personalizzato in un Set.
import java.util.*;

class SetType {
  int i;
  public SetType(int n) { i = n; }
  public boolean equals(Object o) {
    return o instanceof SetType && (i == ((SetType)o).i);
  }
  public String toString() { return Integer.toString(i); }
}

class HashType extends SetType {
  public HashType(int n) { super(n); }
  public int hashCode() { return i; }
}

class TreeType extends SetType
implements Comparable<TreeType> {
  public TreeType(int n) { super(n); }
  public int compareTo(TreeType arg) {
```

```
      return (arg.i < i ? -1 : (arg.i == i ? 0 : 1));
    }
  }

  public class TypesForSets {
    static <T> Set<T> fill(Set<T> set, Class<T> type) {
      try {
        for(int i = 0; i < 10; i++)
            set.add(
              type.getConstructor(int.class).newInstance(i));
      } catch(Exception e) {
        throw new RuntimeException(e);
      }
      return set;
    }
    static <T> void test(Set<T> set, Class<T> type) {
      fill(set, type);
      fill(set, type); // Cerca di aggiungere dei duplicati
      fill(set, type);
      System.out.println(set);
    }
    public static void main(String[] args) {
      test(new HashSet<HashType>(), HashType.class);
      test(new LinkedHashSet<HashType>(), HashType.class);
      test(new TreeSet<TreeType>(), TreeType.class);
      // Operazioni che non funzionano:
      test(new HashSet<SetType>(), SetType.class);
      test(new HashSet<TreeType>(), TreeType.class);
      test(new LinkedHashSet<SetType>(), SetType.class);
      test(new LinkedHashSet<TreeType>(), TreeType.class);
      try {
        test(new TreeSet<SetType>(), SetType.class);
      } catch(Exception e) {
        System.out.println(e.getMessage());
      }
      try {
```

```
      test(new TreeSet<HashType>(), HashType.class);
    } catch(Exception e) {
      System.out.println(e.getMessage());
    }
  }
} /* Output:
[2, 4, 9, 8, 6, 1, 3, 7, 5, 0]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[9, 9, 7, 5, 1, 2, 6, 3, 0, 7, 2, 4, 4, 7, 9, 1, 3, 6, 2, 4,
3, 0, 5, 0, 8, 8, 8, 6, 5, 1]
[0, 5, 5, 6, 5, 0, 3, 1, 9, 8, 4, 2, 3, 9, 7, 3, 4, 4, 0, 7,
1, 9, 6, 2, 1, 8, 2, 8, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
java.lang.ClassCastException: SetType
java.lang.ClassCastException: HashType
*///:~
```

Per dimostrare quali metodi sono necessari per un determinato **Set** e nello stesso tempo evitare la duplicazione di codice, vengono create tre classi. La classe di base, **SetType**, si limita a registrare un **int** che viene prodotto mediante **toString()**. Poiché tutte le classi memorizzate nei **Set** devono avere un **equals()**, questo metodo viene messo a disposizione nella classe di base. L'uguaglianza si basa sul valore **int** di **i**.

La classe **HashType** eredita da **SetType** e fornisce il metodo **hashCode()** necessario per disporre gli oggetti in un'implementazione "hashed" di **Set**.

L'interfaccia **Comparable** implementata da **TreeType** è necessaria per utilizzare un oggetto in qualunque genere di contenitore ordinato, quale un **SortedSet**, di cui **TreeSet** è la sola implementazione. In **compareTo()**, notate che l'autore *non* ha fatto ricorso alla forma "semplice e ovvia" **return i-i2**: questo perché **return i-i2** è un errore di programmazione comune, anche se potrebbe funzionare correttamente se Java gestisse gli **int** privi di segno. Il motivo del possibile mancato funzionamento con gli **int** Java *con segno* è che questi potrebbero non essere abbastanza grandi per rappresentare la differenza di due **int** con segno: non dimenticate infatti che gli **int** vanno da -2.147.483.648 a 2.147.483.647. Per esempio, se **i**

fosse un numero intero positivo particolarmente elevato e **j** un numero intero negativo altrettanto grande, l'operazione **i-j** andrebbe in overflow e restituirebbe un valore negativo, il che equivale a un errore che sarebbe segnalato dal motore Java.

Di norma, vorrete fare in modo che **compareTo()** produca un ordinamento naturale e coerente con il metodo **equals()**: se **equals()** restituisce **true** da un determinato confronto, **compareTo()** dovrebbe fornire un risultato zero per lo stesso confronto; analogamente, se **equals()** restituisce **false** anche **compareTo()** dovrebbe restituire un risultato diverso da zero per quel confronto.

In **TypesForSets** i metodi **fill()** e **test()** sono definiti utilizzando i generici, al fine di evitare codice duplicato. Per verificare il comportamento di un **Set**, **test()** chiama il metodo **fill()** per tre volte sul **set** da testare e cerca di inserirvi alcuni oggetti duplicati. Il metodo **fill()** accetta un **Set** di qualunque tipo e un oggetto **Class** dello stesso tipo; l'oggetto **Class** viene utilizzato per scoprire il costruttore che accetta un argomento **int**, costruttore che verrà poi chiamato per aggiungere gli elementi al **Set**.

Dall'output potete vedere che **HashSet** conserva gli elementi in un ordine "misterioso" (che vi sarà spiegato nel prosieguo del capitolo), **LinkedHashSet** mantiene gli elementi nell'ordine di inserimento, mentre **TreeSet** conserva gli elementi secondo un criterio ordinato: a causa della particolare implementazione di **compareTo()**, questo è l'ordine discendente.

Se cercate di utilizzare tipi che non supportano correttamente le operazioni richieste da un determinato **Set**, andrete incontro a problemi. Se includete in qualsiasi implementazione "hashed" oggetti **SetType** o **TreeType**, che non hanno un metodo **hashCode()** sovrascritto, otterrete valori duplicati, violando di fatto le regole primarie dei **Set**. Questa condizione risulta alquanto irritante, tenuto conto che Java non visualizzerà alcun errore al momento dell'esecuzione.

Tuttavia, il metodo **hashCode()** predefinito è legittimo e questo è un comportamento del tutto coerente, anche se è errato. L'unico modo sicuro per garantire la correttezza operativa in programmi di questo tipo consiste nell'incorporare test unitari nel sistema di build Java: in proposito, consultate il supplemento disponibile all'indirizzo *http://MindView.net/Books/BetterJava.*

Provando a utilizzare un tipo che non implementa **Comparable** in un **TreeSet**, otterrete invece un risultato più indicativo: quando il **TreeSet** cercherà di utilizzare l'oggetto come **Comparable**, il sistema solleverà un'eccezione.

### *SortedSet*

Java garantisce che gli elementi presenti in un **SortedSet** siano ordinati, e questo vi consente di usufruire di altre funzionalità grazie ai metodi disponibili nell'interfaccia **SortedSet**.

**Comparator comparator()**: produce il **Comparator** utilizzato per il **Set** corrente, oppure **null** in caso di ordinamento naturale.

**Object first()**: restituisce l'elemento minimo (il primo).

**Object last()**: restituisce l'elemento massimo (l'ultimo).

**SortedSet subSet(fromElement, toElement)**: produce una vista del **Set** corrente, con gli elementi da **fromElement** incluso fino a **toElement** escluso.

**SortedSet headSet(toElement)**: produce una vista del **Set** corrente, con gli elementi fino a (*inferiori a*) **toElement**.

**SortedSet tailSet(fromElement)**: produce una vista del **Set** corrente, con gli elementi a partire da (*superiori o uguali a*) **fromElement**.

Prendete in esame l'esempio seguente:

```
//: containers/SortedSetDemo.java
// Operazioni ammesse in un TreeSet.
import java.util.*;
import static net.mindview.util.Print.*;
public class SortedSetDemo {
  public static void main(String[] args) {
    SortedSet<String> sortedSet = new TreeSet<String>();
    Collections.addAll(sortedSet,
      "one two three four five six seven eight"
        .split(" "));
    print(sortedSet);
    String low = sortedSet.first();
    String high = sortedSet.last();
    print(low);
    print(high);
    Iterator<String> it = sortedSet.iterator();
    for(int i = 0; i <= 6; i++) {
      if(i == 3) low = it.next();
      if(i == 6) high = it.next();
      else it.next();
```

```
    }
    print(low);
    print(high);
    print(sortedSet.subSet(low, high));
    print(sortedSet.headSet(high));
    print(sortedSet.tailSet(low));
  }
} /* Output:
[eight, five, four, one, seven, six, three, two]
eight
two
one
two
[one, seven, six, three]
[eight, five, four, one, seven, six, three]
[one, seven, six, three, two]
*///:~
```

Tenete presente che **SortedSet** significa “ordinato secondo la funzione di confronto dell’oggetto”, non “in ordine di inserimento”: l’ordine di inserimento può essere mantenuto ricorrendo a un **LinkedHashSet**.

> **Esercizio 9** (2) Utilizzate **RandomGenerator.String** per popolare un **TreeSet**, servendovi però dell’ordinamento alfabetico. Visualizzate il **TreeSet** per verificare il corretto ordinamento.
>
> **Esercizio 10** (7) Utilizzando una **LinkedList** come implementazione di fondo, definite un **SortedSet** personalizzato.

# Code

Tranne le applicazioni di concorrenza, le uniche due implementazioni di **Queue** in Java SE5 sono **LinkedList** e **PriorityQueue**, che si differenziano non tanto nelle prestazioni quanto nel tipo di ordinamento. Di seguito è presentato un esempio di base che illustra la maggior parte delle implementazioni di **Queue** (che non funzioneranno tutte), incluse le code basate sulla concorrenza.

Il meccanismo prevede l’inserimento degli elementi in un’estremità e la loro estrazione dall’altra:

```
//: containers/QueueBehavior.java
// Confronto tra i comportamenti di alcune code
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

public class QueueBehavior {
  private static int count = 10;
  static <T> void test(Queue<T> queue, Generator<T> gen) {
    for(int i = 0; i < count; i++)
      queue.offer(gen.next());
    while(queue.peek() != null)
      System.out.print(queue.remove() + " ");
    System.out.println();
  }
  static class Gen implements Generator<String> {
    String[] s = ("one two three four five six seven " +
      "eight nine ten").split(" ");
    int i;
    public String next() { return s[i++]; }
  }
  public static void main(String[] args) {
    test(new LinkedList<String>(), new Gen());
    test(new PriorityQueue<String>(), new Gen());
    test(new ArrayBlockingQueue<String>(count), new Gen());
    test(new ConcurrentLinkedQueue<String>(), new Gen());
    test(new LinkedBlockingQueue<String>(), new Gen());
    test(new PriorityBlockingQueue<String>(), new Gen());
  }
} /* Output:
one two three four five six seven eight nine ten
eight five four nine one seven six ten three two
one two three four five six seven eight nine ten
one two three four five six seven eight nine ten
one two three four five six seven eight nine ten
eight five four nine one seven six ten three two
```

```
*///:~
```

Noterete che, con l'eccezione delle code con priorità, una **Queue** restituirà gli elementi esattamente nello stesso ordine in cui sono stati inseriti.

## *Code con priorità*

Le code con priorità sono state già presentate brevemente nel Volume 1, Capitolo 11. Un problema molto più interessante è la cosiddetta *to-do-list* (elenco delle priorità), nella quale ogni oggetto contiene una stringa e un valore primario e secondario di priorità. Anche in questo caso, l'ordinamento della lista è gestito mediante un'implementazione di **Comparable**:

```
//: containers/ToDoList.java
// Un utilizzo di PriorityQueue piu' complesso.
import java.util.*;

class ToDoList extends PriorityQueue<ToDoList.ToDoItem> {
  static class ToDoItem implements Comparable<ToDoItem> {
    private char primary;
    private int secondary;
    private String item;
    public ToDoItem(String td, char pri, int sec) {
      primary = pri;
      secondary = sec;
      item = td;
    }
    public int compareTo(ToDoItem arg) {
      if(primary > arg.primary)
        return +1;
      if(primary == arg.primary)
        if(secondary > arg.secondary)
          return +1;
        else if(secondary == arg.secondary)
          return 0;
      return -1;
    }
    public String toString() {
```

```
      return Character.toString(primary) +
        secondary + ": " + item;
    }
  }
  public void add(String td, char pri, int sec) {
    super.add(new ToDoItem(td, pri, sec));
  }
  public static void main(String[] args) {
    ToDoList toDoList = new ToDoList();
    toDoList.add("Empty trash", 'C', 4);
    toDoList.add("Feed dog", 'A', 2);
    toDoList.add("Feed bird", 'B', 7);
    toDoList.add("Mow lawn", 'C', 3);
    toDoList.add("Water lawn", 'A', 1);
    toDoList.add("Feed cat", 'B', 1);
    while(!toDoList.isEmpty())
      System.out.println(toDoList.remove());
  }
} /* Output:
A1: Water lawn
A2: Feed dog
B1: Feed cat
B7: Feed bird
C3: Mow lawn
C4: Empty trash
*///:~
```

Come potete vedere, l'ordinamento delle voci avviene automaticamente grazie alla coda con priorità.

**Esercizio 11** (2) Create una classe contenente un **Integer** che venga inizializzato a un valore tra 0 e 100, utilizzando **java.util.Random**, e implementate l'interfaccia **Comparable** utilizzando questo campo **Integer**. Popolate una **PriorityQueue** con oggetti della vostra classe, poi estraete i valori tramite il metodo **pool()** per mostrare che produce l'ordine previsto.

## Deque

Una *deque* o "coda doppia" funziona esattamente come una normale coda, tuttavia vi permette di aggiungere e rimuovere gli elementi da una qualsiasi delle due estremità. **LinkedList** dispone di metodi che supportano le operazioni di questo tipo di coda, sebbene la libreria standard di Java non abbia un'interfaccia specifica per le code doppie. Di conseguenza, **LinkedList** non può implementare questa interfaccia e non vi permette di eseguire l'upcast a un'interfaccia di **Deque**, come avete fatto con una **Queue** nell'esempio precedente. Potete comunque creare una classe **Deque** servendovi della composizione e semplicemente rendendo pubblici i metodi relativi da **LinkedList**, come nell'esempio seguente:

```
//: net/mindview/util/Deque.java
// Creazione di una Deque da una LinkedList.
package net.mindview.util;
import java.util.*;

public class Deque<T> {
  private LinkedList<T> deque = new LinkedList<T>();
  public void addFirst(T e) { deque.addFirst(e); }
  public void addLast(T e) { deque.addLast(e); }
  public T getFirst() { return deque.getFirst(); }
  public T getLast() { return deque.getLast(); }
  public T removeFirst() { return deque.removeFirst(); }
  public T removeLast() { return deque.removeLast(); }
  public int size() { return deque.size(); }
  public String toString() { return deque.toString(); }
  // E altri metodi eventualmente richiesti...
} ///:~
```

Se includete questa **Deque** in uno dei vostri programmi, potreste scoprire di dovere aggiungere altri metodi per renderla pratica da utilizzare.

Ecco un semplice test della classe **Deque**:

```
//: containers/DequeTest.java
import net.mindview.util.*;
import static net.mindview.util.Print.*;
```

```
public class DequeTest {
  static void fillTest(Deque<Integer> deque) {
    for(int i = 20; i < 27; i++)
      deque.addFirst(i);
    for(int i = 50; i < 55; i++)
      deque.addLast(i);
  }
  public static void main(String[] args) {
    Deque<Integer> di = new Deque<Integer>();
    fillTest(di);
    print(di);
    while(di.size() != 0)
      printnb(di.removeFirst() + " ");
    print();
    fillTest(di);
    while(di.size() != 0)
      printnb(di.removeLast() + " ");
  }
} /* Output:
[26, 25, 24, 23, 22, 21, 20, 50, 51, 52, 53, 54]
26 25 24 23 22 21 20 50 51 52 53 54
54 53 52 51 50 20 21 22 23 24 25 26
*///:~
```

Considerato che è meno probabile che si inseriscano ed estraggano elementi da entrambe le estremità di una coda, **Deque** non è di uso così comune quanto la **Queue**.

## Approfondimenti su Map

Nel Volume 1, Capitolo 11, avete appreso che l'idea di base di una mappa, chiamata anche *array associativo*, è quella di gestire associazioni di chiave-valore (coppie) in modo da recuperare un valore mediante la sua chiave. La libreria standard di Java contiene numerose implementazioni di base di **Map**: **HashMap**, **TreeMap**, **LinkedHashMap**, **WeakHashMap**, **ConcurrentHashMap** e **IdentityHashMap**. Tutte hanno la stessa interfaccia di base di **Map**, ma differiscono per alcuni comportamenti e caratteristiche quali l'efficienza, l'ordine in cui vengono registrate e presentate le coppie, il tempo massimo di

conservazione degli oggetti, le modalità di funzionamento con i programmi multithread e i criteri di determinazione dell'uguaglianza delle chiavi. Del resto, il numero di implementazioni disponibili per **Map** è di per sé indicativo dell'importanza di questo strumento.

Per comprendere a fondo il funzionamento delle **Map**, è importante analizzare il modo in cui è costruito un array associativo. Considerate l'implementazione seguente molto semplice:

```
//: containers/AssociativeArray.java
// Associazione di chiavi e valori.
import static net.mindview.util.Print.*;

public class AssociativeArray<K,V> {
  private Object[][] pairs;
  private int index;
  public AssociativeArray(int length) {
    pairs = new Object[length][2];
  }
  public void put(K key, V value) {
    if(index >= pairs.length)
      throw new ArrayIndexOutOfBoundsException();
    pairs[index++] = new Object[]{ key, value };
  }
  @SuppressWarnings("unchecked")
  public V get(K key) {
    for(int i = 0; i < index; i++)
      if(key.equals(pairs[i][0]))
        return (V)pairs[i][1];
    return null; // Chiave non trovata
  }
  public String toString() {
    StringBuilder result = new StringBuilder();
    for(int i = 0; i < index; i++) {
      result.append(pairs[i][0].toString());
      result.append(" : ");
      result.append(pairs[i][1].toString());
      if(i < index - 1)
```

```
        result.append("\n");
      }
      return result.toString();
    }
    public static void main(String[] args) {
      AssociativeArray<String,String> map =
        new AssociativeArray<String,String>(6);
     map.put("sky", "blue");
     map.put("grass", "green");
     map.put("ocean", "dancing");
     map.put("tree", "tall");
     map.put("earth", "brown");
     map.put("sun", "warm");
      try {
        map.put("extra", "object"); // Supera la fine
      } catch(ArrayIndexOutOfBoundsException e) {
        print("Too many objects!");
      }
      print(map);
      print(map.get("ocean"));
    }
  } /* Output:
  Too many objects!
  sky : blue
  grass : green
  ocean : dancing
  tree : tall
  earth : brown
  sun : warm
  dancing
  *///:~
```

I metodi essenziali in un array associativo sono **put()** e **get()**, ma per semplificare ulteriormente la visualizzazione si è scelto di sovrascrivere il metodo **toString()** affinché visualizzi le coppie di chiave e valore. Per mostrare il funzionamento del programma, **main()** carica un **AssociativeArray** con le coppie di stringhe e visualizza la mappa risultante, seguita da una **get()** di uno dei valori.

Per utilizzare il metodo **get()** passate la **key** da ricercare: di ritorno otterrete il valore associato o **null**, se questo non è stato trovato. Il metodo **get()** si serve di quello che probabilmente è il criterio meno efficiente per individuare il valore, cominciando dall'inizio dell'array e confrontando le chiavi con **equals()**: tenete comunque presente che in questo caso è importante la semplicità, non l'efficienza.

Per quanto possa essere istruttiva, tuttavia, questa versione di **Map** non è molto efficiente e per di più ha dimensioni fisse, il che la rende per nulla flessibile. Fortunatamente, le **Map in java.util** non presentano questi problemi e possono essere sostituite nel suddetto esempio.

> **Esercizio 12** (1) Sostituite una **HashMap**, una **TreeMap** e una **LinkedHashMap** nel **main()** di **AssociativeArray.java**.
>
> **Esercizio 13** (4) Servitevi di **AssociativeArray.java** per creare un programma di conteggio delle occorrenze di parole, facendo corrispondere **String** a **Integer**. Servendovi dell'utility **net.mindview.util.TextFile** di questo volume, aprite un file di testo e classificate le parole contenute tenendo conto di spazi e punteggiatura, contando il numero di occorrenze per ciascuna parola.

## *Prestazioni*

Le prestazioni sono una questione fondamentale per le mappe, nelle quali il recupero delle chiavi mediante ricerca lineare con **get()** risulta particolarmente lento. È in queste occasioni che si rivela utile **HashMap**, accelerando le operazioni: anziché ricercare le chiavi in modo sequenziale, questa classe si serve di un valore speciale chiamato *hash code* o *codice hash*. L'utilizzo del codice hash è una tecnica che consente di "intercettare" alcune informazioni su un determinato oggetto per trasformarle in un **int** "relativamente univoco" per quell'oggetto; **hashCode()** è un metodo della classe radice **Object**, grazie al quale tutti gli oggetti Java possono produrre un codice hash. Una **HashMap** può così prendere il metodo **hashCode()** dell'oggetto e utilizzarlo per cercare rapidamente la chiave, con un beneficio enorme in termini di prestazioni.[6]

6. Se questo miglioramento di prestazioni non dovesse ancora soddisfarvi, potete accelerare ulteriormente la consultazione di una tabella scrivendo la vostra **Map** e personalizzandola ai vostri tipi specifici, per eliminare i ritardi dovuti al downcasting e all'upcasting nei confronti di **Object**. Al fine di ottenere livelli prestazionali ancora più elevati, i cultori della performance possono consultare *The Art of Computer Programming, Volume 3: Sorting and Searching, seconda edizione*, per sostituire con gli array le liste con *overflow bucket*; gli array, infatti, offrono due benefici supplementari: possono essere ottimizzati in funzione delle caratteristiche dei dischi e fare risparmiare moltissimo tempo nella creazione e successiva eliminazione di record singoli.

Di seguito sono elencate le implementazioni **Map** di base; l'asterisco in corrispondenza di **HashMap** indica che, in assenza di altri vincoli, questa dovrebbe essere la scelta predefinita perché è ottimizzato in termini di velocità: le altre varianti danno risalto ad altre caratteristiche e non sono così veloci quanto **HashMap**.

| | |
|---|---|
| **HashMap*** | Implementazione basata su una tabella hash, da utilizzare in sostituzione di **Hashtable**: offre prestazioni costanti nel tempo per quanto concerne l'inserimento e l'individuazione delle coppie. Le prestazioni possono essere regolate tramite costruttori che consentono l'impostazione della *capienza* (*capacity*) e del *coefficiente di carico* (*load factor*) della tabella hash. |
| **LinkedHashMap** | Funziona come una **HashMap**, ma quando la iterate ottenete le coppie in ordine di inserimento, o nell'ordine utilizzato meno di recente (*LRU*, *Least Recently Used*). Questa classe è un po' più lenta di una **HashMap**, tranne in fase di iterazione, dove si rivela più rapida grazie alla lista collegata utilizzata per mantenere l'ordinamento interno. |
| **TreeMap** | Implementazione basata sulla cosiddetta "alberatura rossonera", una struttura dati ad albero in cui ciascun nodo è associato ai colori rosso o nero, secondo regole appropriate. Lo scopo di una **TreeMap** è ottenere risultati ordinati, e la visualizzazione delle chiavi o delle coppie avviene secondo il criterio di ordinamento determinato da **Comparable** o da **Comparator**. **TreeMap** è l'unica **Map** con il metodo **subMap()**, che consente di restituire una porzione dell'alberatura. |
| **WeakHashMap** | Implementazione basata su una tabella hash, con "chiavi deboli" (*weak key*). Una voce di una **WeakHashMap** viene automaticamente rimossa quando la chiave relativa non viene più utilizzata. Più esattamente, la presenza di una mappatura per una determinata chiave non impedisce l'eliminazione della chiave tramite la garbage collection. Quando una chiave è stata eliminata la voce relativa viene rimossa dalla mappa, pertanto il comportamento di questa classe è diverso da quello di altre implementazioni di **Map**. |

| **ConcurrentHashMap** | Una **Map** con sicurezza di thread la quale non implica lock di sincronizzazione, che verrà esaminata nel Volume 3, Capitolo 1. |
|---|---|
| **IdentityHashMap** | Una mappa hash che per il confronto delle chiavi utilizza l'operatore **==** anziché il metodo **equals()**. Questa variante non è di uso generale, ma riservata alla risoluzione di particolari problemi. |

L'algoritmo di *hashing* (o *hash*) è il criterio più comunemente adottato per memorizzare gli elementi in una mappa: vedrete in seguito i dettagli del suo funzionamento.

I requisiti delle chiavi utilizzate in una **Map** sono gli stessi degli elementi in un **Set**, dimostrati nel programma **TypesForSets.java**. Ogni chiave deve avere un metodo **equals()**; inoltre, se la chiave è utilizzata in una **Map** di tipo hash, deve anche avere un metodo **hashCode()** e, se la chiave è adottata in una **TreeMap**, deve implementare **Comparable**.

L'esempio seguente mostra le operazioni rese disponibili dall'interfaccia di **Map**, utilizzando il dataset di prova **CountingMapData** definito in precedenza:

```
//: containers/Maps.java
// Operazioni ammesse nelle Map.
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class Maps {
  public static void printKeys(Map<Integer,String> map) {
    printnb("Size = " + map.size() + ", ");
    printnb("Keys: ");
    print(map.keySet()); // Produce un Set delle chiavi
  }
  public static void test(Map<Integer,String> map) {
    print(map.getClass().getSimpleName());
    map.putAll(new CountingMapData(25));
    // Le chiavi di Map hanno lo stesso comportamento che in
    // 'Set':
    map.putAll(new CountingMapData(25));
```

```
    printKeys(map);
    // Produce una Collection di valori:
    printnb("Values: ");
    print(map.values());
    print(map);
    print("map.containsKey(11): " + map.containsKey(11));
    print("map.get(11): " + map.get(11));
    print("map.containsValue(\"F0\"): "
      + map.containsValue("F0"));
    Integer key = map.keySet().iterator().next();
    print("First key in map: " + key);
    map.remove(key);
    printKeys(map);
    map.clear();
    print("map.isEmpty(): " + map.isEmpty());
    map.putAll(new CountingMapData(25));
    // Le operazioni su Set modificano la Map:
    map.keySet().removeAll(map.keySet());
    print("map.isEmpty(): " + map.isEmpty());
  }
  public static void main(String[] args) {
    test(new HashMap<Integer,String>());
    test(new TreeMap<Integer,String>());
    test(new LinkedHashMap<Integer,String>());
    test(new IdentityHashMap<Integer,String>());
    test(new ConcurrentHashMap<Integer,String>());
    test(new WeakHashMap<Integer,String>());
  }
} /* Output:
HashMap
Size = 25, Keys: [15, 8, 23, 16, 7, 22, 9, 21, 6, 1, 14, 24,
4, 19, 11, 18, 3, 12, 17, 2, 13, 20, 10, 5, 0]
Values: [P0, I0, X0, Q0, H0, W0, J0, V0, G0, B0, O0, Y0, E0,
T0, L0, S0, D0, M0, R0, C0, N0, U0, K0, F0, A0]
{15=P0, 8=I0, 23=X0, 16=Q0, 7=H0, 22=W0, 9=J0, 21=V0, 6=G0,
1=B0, 14=O0, 24=Y0, 4=E0, 19=T0, 11=L0, 18=S0, 3=D0, 12=M0,
17=R0, 2=C0, 13=N0, 20=U0, 10=K0, 5=F0, 0=A0}
```

```
map.containsKey(11): true
map.get(11): L0
map.containsValue("F0"): true
First key in map: 15
Size = 24, Keys: [8, 23, 16, 7, 22, 9, 21, 6, 1, 14, 24, 4,
19, 11, 18, 3, 12, 17, 2, 13, 20, 10, 5, 0]
map.isEmpty(): true
map.isEmpty(): true
...
*///:~
```

Il metodo **printKeys()** dimostra come produrre una vista di tipo **Collection** per una **Map**. Il metodo **keySet()** produce un **Set** supportato dalle chiavi nella **Map**. Grazie ai miglioramenti ottenuti con Java SE5 per quanto concerne la visualizzazione e la stampa, potete visualizzare semplicemente il risultato del metodo **values()**, che produce una **Collection** contenente tutti i valori presenti nella **Map**: ricordate infatti che, benché le chiavi debbano essere univoche, i valori possono contenere duplicati. Dal momento che queste **Collection** sono supportate da una **Map**, qualsiasi cambiamento operato in una **Collection** si rifletterà nella **Map** a essa associata.

Il resto del programma fornisce semplici esempi di tutte le funzionalità di **Map** e verifica ogni tipo di base di **Map**.

**Esercizio 14** (3) Dimostrate che **java.util.Properties** funziona nel programma precedente.

## *SortedMap*

Una **SortedMap**, di cui la **TreeMap** è l'unica implementazione disponibile, assicura che le chiavi siano mantenute secondo un criterio di ordinamento, mediante il quale si rendono disponibili altre funzionalità tramite i metodi dell'interfaccia **SortedMap**.

**Comparator comparator()**: produce il comparatore utilizzato per la **Map** corrente, oppure **null** in caso di ordinamento naturale.

**T firstKey()**: restituisce la chiave minima (la prima).

**T lastKey()**: restituisce la chiave massima (l'ultima).

**SortedMap subMap(fromKey, toKey)**: produce una vista della **Map** corrente, le cui chiavi vanno da **fromKey** compresa fino a **toKey** esclusa.

**SortedMap headMap(toKey)**: produce una vista della **Map** corrente, le cui chiavi sono inferiori a **toKey**.

**SortedMap tailMap(fromKey)**: produce una vista della **Map** corrente, le cui chiavi sono superiori o uguali a **fromKey**.

L'esempio seguente è simile a **SortedSetDemo.java** e mostra i comportamenti di **TreeMap**:

```
//: containers/SortedMapDemo.java
// Operazioni ammesse in una TreeMap.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class SortedMapDemo {
  public static void main(String[] args) {
    TreeMap<Integer,String> sortedMap =
      new TreeMap<Integer,String>(new CountingMapData(10));
    print(sortedMap);
    Integer low = sortedMap.firstKey();
    Integer high = sortedMap.lastKey();
    print(low);
    print(high);
    Iterator<Integer> it = sortedMap.keySet().iterator();
    for(int i = 0; i <= 6; i++) {
      if(i == 3) low = it.next();
      if(i == 6) high = it.next();
      else it.next();
    }
    print(low);
    print(high);
    print(sortedMap.subMap(low, high));
    print(sortedMap.headMap(high));
    print(sortedMap.tailMap(low));
  }
} /* Output:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0, 9=J0}
0
```

```
9
3
7
{3=D0, 4=E0, 5=F0, 6=G0}
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0}
{3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0, 9=J0}
*///:~
```

In questo codice, le coppie chiave-valore vengono memorizzate in ordine di chiave. Dal momento che **TreeMap** ha un criterio di ordinamento implicito, il concetto di "posizione" ha perfettamente senso, per consentirvi di ottenere il primo e l'ultimo elemento e alcune sottomappe.

## LinkedHashMap

Per velocizzare le operazioni, **LinkedHashMap** sottopone tutto ad hash e produce le chiavi in ordine di inserimento durante un'iterazione: **System.out.println()** itera la mappa, permettendo di visualizzare i risultati. In più, il costruttore di una **LinkedHashMap** può essere configurato per utilizzare l'algoritmo LRU (*Least Recently Used*) basato sugli accessi meno recenti, in modo che gli elementi che non sono stati utilizzati (e che pertanto sono candidati alla rimozione) compaiano all'inizio della lista: questo vi consente di creare programmi che periodicamente eseguano il cleanup per risparmiare spazio su disco. Ecco un semplice esempio che mostra entrambe le funzionalità:

```
//: containers/LinkedHashMapDemo.java
// Operazioni ammesse in una LinkedHashMap.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class LinkedHashMapDemo {
  public static void main(String[] args) {
    LinkedHashMap<Integer,String> linkedMap =
      new LinkedHashMap<Integer,String>(
        new CountingMapData(9));
    print(linkedMap);
    // Ordine Least-recently-used:
    linkedMap =
```

```
      new LinkedHashMap<Integer,String>(16, 0.75f, true);
    linkedMap.putAll(new CountingMapData(9));
    print(linkedMap);
    for(int i = 0; i < 6; i++) // Gestisce gli accessi
      linkedMap.get(i);
    print(linkedMap);
    linkedMap.get(0);
    print(linkedMap);
  }
} /* Output:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0}
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0}
{6=G0, 7=H0, 8=I0, 0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0}
{6=G0, 7=H0, 8=I0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 0=A0}
*///:~
```

L'output mostra che le coppie vengono effettivamente "attraversate" in ordine di inserimento, anche per la versione LRU. Tuttavia, dopo aver avuto accesso alle (sole) primi sei voci nella versione LRU, le ultime tre voci si spostano all'inizio dell'elenco: in seguito, quando "**0**" viene nuovamente raggiunto si sposta alla fine dell'elenco.

## Hashing e codici hash

Negli esempi del Volume 1, Capitolo 11, come chiavi di **HashMap** sono state utilizzate classi predefinite. Questi esempi funzionano perché le classi predefinite dispongono di tutta l'infrastruttura necessaria per comportarsi correttamente come chiavi.

Se create le vostre classi da utilizzare come chiavi **HashMaps** e dimenticate di includere questa infrastruttura, si verificheranno alcuni inconvenienti. Considerate, per esempio, un sistema di previsione meteorologica che faccia corrispondere oggetti **Groundhog** con oggetti **Prediction**. Il problema sembra piuttosto semplice, basato sulla creazione di due classi e sull'utilizzo di **Groundhog** come chiave e di **Prediction** come valore:

```
//: containers/Groundhog.java
// All'apparenza e' plausibile, ma non funziona come chiave
// HashMap.
```

```
public class Groundhog {
  protected int number;
  public Groundhog(int n) { number = n; }
  public String toString() {
    return "Groundhog #" + number;
  }
} ///:~

//: containers/Prediction.java
// Prevede il tempo con le marmotte (Groundhog).
import java.util.*;

public class Prediction {
  private static Random rand = new Random(47);
  private boolean shadow = rand.nextDouble() > 0.5;
  public String toString() {
    if(shadow)
      return "Six more weeks of Winter!";
    else
      return "Early Spring!";
  }
} ///:~

//: containers/SpringDetector.java
// Che tempo fara'?
import java.lang.reflect.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class SpringDetector {
  // Usa Groundhog oppure una classe derivata da Groundhog:
  public static <T extends Groundhog>
  void detectSpring(Class<T> type) throws Exception {
    Constructor<T> ghog = type.getConstructor(int.class);
    Map<Groundhog,Prediction> map =
      new HashMap<Groundhog,Prediction>();
```

```
    for(int i = 0; i < 10; i++)
      map.put(ghog.newInstance(i), new Prediction());
    print("map = " + map);
    Groundhog gh = ghog.newInstance(3);
    print("Looking up prediction for " + gh);
    if(map.containsKey(gh))
      print(map.get(gh));
    else
      print("Key not found: " + gh);
  }
  public static void main(String[] args) throws Exception {
    detectSpring(Groundhog.class);
  }
} /* Output:
map = {Groundhog #3=Early Spring!, Groundhog #7=Early
Spring!, Groundhog #5=Early Spring!, Groundhog #9=Six more
weeks of Winter!, Groundhog #8=Six more weeks of Winter!,
Groundhog #0=Six more weeks of Winter!, Groundhog #6=Early
Spring!, Groundhog #4=Six more weeks of Winter!, Groundhog
#1=Six more weeks of Winter!, Groundhog #2=Early Spring!}
Looking up prediction for Groundhog #3
Key not found: Groundhog #3
*///:~
```

In questo listato, a ogni marmotta (**Groundhog**) è assegnato un codice identificativo: in questo modo, potete consultare una previsione (**Prediction**) in **HashMap** semplicemente richiedendo quella associata a una determinata **Groundhog**. La classe **Prediction** contiene un valore **boolean** che viene inizializzato utilizzando **java.util.random()**, e un metodo **toString()** per interpretare il risultato. Il metodo **detectSpring()** viene creato mediante la riflessione, per istanziare e utilizzare la **class Groundhog** o qualunque classe derivata da essa. Questo metodo risulterà utile tra breve, quando erediterete un nuovo tipo di **Groundhog** per risolvere il problema dimostrato di seguito.

Una **HashMap** viene popolata con le **Groundhog** e le **Prediction** associate; la **HashMap** viene poi visualizzata per consentirvi di esaminare in che modo è stata popolata la mappa. In seguito, la **Groundhog** con il codice identificativo 3 è utilizzato come chiave per recuperare la previsione **Groundhog** numero 3, che come vedete deve esistere nella **Map**.

Sembra abbastanza semplice, ma non funziona: il programma non riesce a trovare la chiave numero 3. Il problema è che la classe **Groundhog** eredita automaticamente dalla classe radice **Object**, ed è quindi il metodo **hashCode** di **Object()** a essere utilizzato per generare il codice hash di ogni oggetto. In modalità predefinita, questo metodo si serve soltanto dell'indirizzo del relativo oggetto: pertanto la prima istanza di **Groundhog(3)** *non* produce un codice hash uguale a quello della seconda istanza di **Groundhog(3)**, vale a dire quella che volete utilizzare per la ricerca.

Potreste pensare che basti sovrascrivere opportunamente il metodo **hashCode()**. Ma non è sufficiente: dovrete anche sovrascrivere il metodo **equals()**, che è anch'esso parte di **Object**; questo metodo viene utilizzato da **HashMap** per determinare se la chiave corrente è uguale a una qualsiasi delle chiavi presenti nella tabella.

Un metodo **equals()** ben scritto deve soddisfare cinque condizioni, elencate di seguito.

1. Essere *riflessivo*: per qualsiasi **x**, **x.equals(x)** dovrebbe restituire **true**.
2. Essere *simmetrico*: per qualsiasi **x** e **y**, **x.equals(y)** dovrebbe restituire **true** se e soltanto se **y.equals(x)** restituisce **true**.
3. Essere *transitivo*: per qualsiasi **x**, **y** e **z**, se **x.equals(y)** e **y.equals(z)** restituiscono entrambi **true**, anche **x.equals(z)** dovrebbe restituire **true**.
4. Essere *costante*: per qualsiasi **x** e **y**, ripetute chiamate di **x.equals(y)** dovrebbero coerentemente restituire **true** o **false**, sempre che non vengano modificate le informazioni utilizzate nelle valutazioni di uguaglianza.
5. Per qualsiasi **x** non nulla, **x.equals(null)** dovrebbe restituire **false**.

Il metodo predefinito **Object.equals()** si limita a confrontare gli indirizzi dell'oggetto, pertanto un **Groundhog(3)** non sarà uguale a un altro **Groundhog(3)**. Di conseguenza, per utilizzare le vostre classi in una **HashMap** dovete sovrascrivere sia **hashCode()** sia **equals()**, secondo le indicazioni fornite dal seguente codice, che risolve il "problema delle marmotte":

```
//: containers/Groundhog2.java
// Una classe che viene usata come chiave in una HashMap
// deve sovrascrivere hashCode() ed equals().

public class Groundhog2 extends Groundhog {
  public Groundhog2(int n) { super(n); }
  public int hashCode() { return number; }
  public boolean equals(Object o) {
```

```
    return o instanceof Groundhog2 &&
      (number == ((Groundhog2)o).number);
  }
} ///:~

//: containers/SpringDetector2.java
// Questa volta la chiave funziona.

public class SpringDetector2 {
  public static void main(String[] args) throws Exception {
    SpringDetector.detectSpring(Groundhog2.class);
  }
} /* Output:
map = {Groundhog #2=Early Spring!, Groundhog #4=Six more
weeks of Winter!, Groundhog #9=Six more weeks of Winter!,
Groundhog #8=Six more weeks of Winter!, Groundhog #6=Early
Spring!, Groundhog #1=Six more weeks of Winter!, Groundhog
#3=Early Spring!, Groundhog #7=Early Spring!, Groundhog
#5=Early Spring!, Groundhog #0=Six more weeks of Winter!}
Looking up prediction for Groundhog #3
Early Spring!*///:~
```

**Groundhog2.hashCode()** restituisce il numero della marmotta sotto forma di valore hash: in questo caso specifico, il programmatore ha la responsabilità di accertarsi che non esistano due marmotte con lo stesso codice identificativo. Come vedrete meglio nel prosieguo del capitolo, non è necessario che **hashCode()** restituisca un identificativo univoco, tuttavia il metodo **equals()** deve determinare con rigore se due oggetti sono equivalenti. In questo esempio **equals()** si basa sul numero identificativo della marmotta (**number**), quindi se esistono due oggetti **Groundhog2** come chiavi di **HashMap** con lo stesso numero di marmotta, il meccanismo non funzionerà.

Benché sembri che il metodo **equals()** stia soltanto verificando se l'argomento sia un'istanza di **Groundhog2** (utilizzando la parola chiave **instanceof**, descritta nel Capitolo 2), in realtà **instanceof** esegue in autonomia un secondo controllo di integrità per vedere se l'oggetto è **null**, dal momento che **instanceof** produce **false** se l'argomento sul lato sinistro della dichiarazione è **null**. Supponendo che l'elemento sia del tipo corretto e non sia **null**, il confronto si

baserà sui valori **number** effettivi di ogni oggetto: l'output dimostra che ora il comportamento è corretto.

Nel creare la vostra classe personalizzata in un **HashSet**, dovete prestare attenzione agli stessi problemi che potrebbero verificarsi quando la vostra classe è utilizzata come chiave in un **HashMap**.

## Approfondimenti su hashCode()

L'esempio precedente è soltanto un passo nella direzione della soluzione corretta al problema e dimostra che, se non eseguite l'override di **hashCode()** ed **equals()** per la vostra chiave, la struttura dati "hashed" (sia essa **HashSet**, **HashMap**, **LinkedHashSet** o **LinkedHashMap**) difficilmente gestirà la vostra chiave in modo corretto. Per realizzare una *buona* soluzione al problema, tuttavia, occorre comprendere come funziona internamente una struttura dati con hash.

Innanzitutto considerate la motivazione su cui si fonda l'algoritmo di hashing: la necessità di recuperare un oggetto servendosi di un altro oggetto. Questo risultato potrebbe tuttavia essere ottenuto anche con una **TreeMap** o persino con una **Map** personalizzata.

Diversamente da quanto avviene in un'implementazione con hash, l'esempio seguente implementa una **Map** servendosi di una coppia di **ArrayList**; a differenza di **AssociativeArray.java**, però, il codice include un'implementazione completa dell'interfaccia di **Map**, che tiene conto del metodo **entrySet()**:

```
//: containers/SlowMap.java
// Una Map implementata con ArrayList.
import java.util.*;
import net.mindview.util.*;

public class SlowMap<K,V> extends AbstractMap<K,V> {
  private List<K> keys = new ArrayList<K>();
  private List<V> values = new ArrayList<V>();
  public V put(K key, V value) {
    V oldValue = get(key); // Il vecchio valore oppure null
    if(!keys.contains(key)) {
      keys.add(key);
      values.add(value);
    } else
```

```
      values.set(keys.indexOf(key), value);
    return oldValue;
  }
  public V get(Object key) { // key e' di tipo Object, non K
    if(!keys.contains(key))
      return null;
    return values.get(keys.indexOf(key));
  }
  public Set<Map.Entry<K,V>> entrySet() {
    Set<Map.Entry<K,V>> set= new HashSet<Map.Entry<K,V>>();
    Iterator<K> ki = keys.iterator();
    Iterator<V> vi = values.iterator();
    while(ki.hasNext())
      set.add(new MapEntry<K,V>(ki.next(), vi.next()));
    return set;
  }
  public static void main(String[] args) {
    SlowMap<String,String> m= new SlowMap<String,String>();
    m.putAll(Countries.capitals(15));
    System.out.println(m);
    System.out.println(m.get("EGYPT"));
    System.out.println(m.entrySet());
  }
} /* Output:
{CAMEROON=Yaounde, CONGO=Brazzaville, CAPE VERDE=Praia,
ALGERIA=Algiers, COMOROS=Moroni, CENTRAL AFRICAN
REPUBLIC=Bangui, EQUATORIAL GUINEA=Malabo, BOTSWANA=Gaberone,
BURUNDI=Bujumbura, BENIN=Porto-Novo, EGYPT=Cairo,
ANGOLA=Luanda, BURKINA FASO=Ouagadougou, DJIBOUTI=Dijibouti,
CHAD=N'djamena}
Cairo
[CAMEROON=Yaounde, CONGO=Brazzaville, CAPE VERDE=Praia,
ALGERIA=Algiers, COMOROS=Moroni, CENTRAL AFRICAN
REPUBLIC=Bangui, EQUATORIAL GUINEA=Malabo, BOTSWANA=Gaberone,
BURUNDI=Bujumbura, BENIN=Porto-Novo, EGYPT=Cairo,
ANGOLA=Luanda, BURKINA FASO=Ouagadougou, DJIBOUTI=Dijibouti,
CHAD=N'djamena]
*///:~
```

Il metodo **put()** dispone semplicemente le chiavi e i valori negli **ArrayLists** corrispondenti, e in conformità con l'interfaccia **Map** deve restituire la vecchia chiave oppure **null**, in mancanza di una chiave esistente. Sempre secondo le specifiche di **Map**, **get()** restituisce **null** se la chiave non è nella **SlowMap**: qualora la chiave esista, viene utilizzata per recuperare l'indice numerico che indica la relativa posizione nella **List keys**, e questo numero è adottato come indice per produrre il valore collegato da **List values**. Notate che il tipo di **key** in **get()** è **Object**, invece del tipo parametrizzato **K** che potreste aspettarvi e che è stato utilizzato in **AssociativeArray.java**. Questo è un tipico risultato dell'introduzione a posteriori dei generici nel linguaggio Java: se fossero stati una funzionalità originale del linguaggio, in **get()** si sarebbe potuto indicare il tipo del relativo parametro.

Il metodo **Map.entrySet()** deve produrre un insieme di oggetti **Map.Entry**. Tuttavia, **Map.Entry** è un'interfaccia che descrive una struttura dipendente dall'implementazione, quindi se volete realizzare il vostro tipo di **Map** dovrete anche definire un'implementazione di **Map.Entry**:

```
//: containers/MapEntry.java
// Una semplice Map.Entry per implementazioni Map d'esempio.
import java.util.*;
public class MapEntry<K,V> implements Map.Entry<K,V> {
  private K key;
  private V value;
  public MapEntry(K key, V value) {
    this.key = key;
    this.value = value;
  }
  public K getKey() { return key; }
  public V getValue() { return value; }
  public V setValue(V v) {
    V result = value;
    value = v;
    return result;
  }
  public int hashCode() {
    return (key==null ? 0 : key.hashCode()) ^
      (value==null ? 0 : value.hashCode());
  }
  public boolean equals(Object o) {
```

```
      if(!(o instanceof MapEntry)) return false;
      MapEntry me = (MapEntry)o;
      return
        (key == null ?
         me.getKey() == null : key.equals(me.getKey())) &&
        (value == null ?
         me.getValue()== null : value.equals(me.getValue()));
    }
    public String toString() { return key + "=" + value; }
  } ///:~
```

In questo codice, una classe molto semplice chiamata **MapEntry** memorizza e recupera le chiavi e i valori, utilizzati in **entrySet()** per produrre un **Set** di coppie chiave-valore. Osservate che **entrySet()** si serve di un **HashSet** per conservare le coppie e che **MapEntry** adotta l'approccio più facile, limitandosi a utilizzare il metodo **hashCode()** di **key**. Benché questa soluzione sia elementare e sembri funzionare nel banalissimo test in **SlowMap.main()**, non è un'implementazione corretta perché esegue una copia delle chiavi e dei valori. Un'implementazione corretta di **entrySet()**, infatti, fornirà una *vista* di **Map** anziché una copia, e consentirà di modificare la mappa originale, operazione impossibile con una copia. L'Esercizio 16 vi offre la possibilità di ovviare al problema.

Ricordate che il metodo **equals()** in **MapEntry** deve controllare sia le chiavi sia i valori. Il funzionamento del metodo **hashCode()**, invece, sarà descritto tra breve.

La rappresentazione **String** del contenuto di **SlowMap** viene prodotta automaticamente dal metodo **toString()** definito in **AbstractMap**.

In **SlowMap.main()** viene caricata una **SlowMap**, di cui in seguito viene visualizzato il contenuto: una chiamata a **get()** dimostra il corretto funzionamento.

**Esercizio 15** (1) Ripetete l'Esercizio 13 utilizzando una **SlowMap**.

**Esercizio 16** (7) Applicate i test di **Maps.java** a **SlowMap** per verificarne il funzionamento: eseguite le correzioni opportune per qualunque cosa non funzioni in **SlowMap**.

**Esercizio 17** (2) Implementate il resto dell'interfaccia **Map** per **SlowMap**.

**Esercizio 18** (3) Basandovi su **SlowMap.java**, create uno **SlowSet**.

## Ottimizzazione dell'hashing

Nel codice di **SlowMap.java** avete visto che non è poi molto difficile realizzare un nuovo tipo di **Map**; tuttavia, una **SlowMap** non è molto veloce, pertanto probabilmente non ve ne servireste avendo a disposizione un'alternativa. Il problema risiede nella ricerca della chiave: dato che le chiavi non sono conservate in nessun ordine particolare, viene utilizzata una semplice ricerca lineare, vale a dire la tecnica più lenta che si possa adottare per qualunque ricerca.

Lo scopo primario di un algoritmo di hashing è migliorare la velocità di esecuzione, in questo caso delle ricerche. Poiché il collo di bottiglia è appunto la velocità di ricerca della chiave, una delle soluzioni possibili è mantenere le chiavi ordinate e servirsi di **Collections.binarySearch()** per eseguire la ricerca. A questo proposito, in uno dei successivi esempi vedrete in dettaglio come avviene tale processo.

La tecnica di hashing si spinge oltre, arrivando a fare in modo che dobbiate semplicemente registrare la chiave *da qualche parte*, cosicché possa essere recuperata in maniera rapida. Come sapete, la struttura più veloce in cui registrare un gruppo di elementi è l'array, pertanto vi servirete di questo oggetto per rappresentare le informazioni sulla chiave: notate che si è parlato di "informazioni sulla chiave", non della chiave stessa. Un array non è tuttavia ridimensionabile, ed ecco quindi che si presenta un problema: come fare per memorizzare un numero arbitrario di valori in una **Map**, se il numero di chiavi è determinato dalle dimensioni fisse di un array?

La risposta a questa domanda è che l'array non conterrà le chiavi: dall'oggetto chiave verrà derivato un numero che sarà utilizzato come indice nell'array. Questo numero è il *codice hash*, prodotto dal metodo **hashCode()** (la *funzione di hash*, in termini informatici) definito in **Object** e presumibilmente sovrascritto nella vostra classe.

Per risolvere il problema dell'array, che ha dimensione fissa, occorre normalizzare il valore dell'hash alla dimensione dell'array. In questo modo, è possibile che più chiavi producano lo stesso indice: si parlerà allora di *collisioni*. Per questo motivo non ha importanza quanto sia grande l'array, perché il codice hash di qualsiasi oggetto chiave corrisponderà comunque a qualche elemento dell'array.

Quindi, il processo di recupero di un valore inizia con il calcolo del codice hash, che viene poi indicizzato nell'array. Se potete garantire che non vi saranno collisioni (eventualità possibile soltanto nel caso di un numero di valori fisso), allora avete una *funzione di hash perfetta*, un evento molto raro.[7]

---

7. In Java SE5 la funzione di hashing perfetta è implementata in **EnumMap** ed **EnumSet**, perché una **enum** definisce un numero fisso di istanze; si veda in proposito il Capitolo 7.

In tutti gli altri casi, le collisioni vengono gestite mediante il cosiddetto *concatenamento esterno* (*external chaining*): l'array non fa riferimento direttamente a un valore ma a un elenco di valori, che vengono cercati in modo sequenziale con il metodo **equals()**. Naturalmente questa funzione di ricerca è molto più lenta, ma se la funzione di hash è ottimale in ogni posizione vi saranno soltanto pochi valori; pertanto, anziché eseguire la ricerca in tutto l'elenco, passerete rapidamente a una posizione in cui dovrete confrontare soltanto alcune voci per trovare il valore richiesto. Questo meccanismo è molto più veloce, e ciò spiega perché la **HashMap** è così rapida.

Ora che conoscete i principi fondamentali dell'hashing, potete implementare una semplice **Map** di tipo hash:

```
//: containers/SimpleHashMap.java
// Una Map dimostrativa di tipo hash.
import java.util.*;
import net.mindview.util.*;

public class SimpleHashMap<K,V> extends AbstractMap<K,V> {
  // Sceglie un numero primo per le dimensioni della tabella
  // di hash, al fine di ottenere una distribuzione uniforme:
  static final int SIZE = 997;
  // Non e' possibile avere un array fisico di generici,
  // tuttavia e' possibile ottenerne uno per upcasting:
  @SuppressWarnings("unchecked")
  LinkedList<MapEntry<K,V>>[] buckets =
    new LinkedList[SIZE];
  public V put(K key, V value) {
    V oldValue = null;
    int index = Math.abs(key.hashCode()) % SIZE;
    if(buckets[index] == null)
      buckets[index] = new LinkedList<MapEntry<K,V>>();
    LinkedList<MapEntry<K,V>> bucket = buckets[index];
    MapEntry<K,V> pair = new MapEntry<K,V>(key, value);
    boolean found = false;
    ListIterator<MapEntry<K,V>> it = bucket.listIterator();
    while(it.hasNext()) {
      MapEntry<K,V> iPair = it.next();
      if(iPair.getKey().equals(key)) {
```

```
        oldValue = iPair.getValue();
        it.set(pair); // Sostituisce i vecchi valori con i nuovi
        found = true;
        break;
      }
    }
    if(!found)
      buckets[index].add(pair);
    return oldValue;
  }
  public V get(Object key) {
    int index = Math.abs(key.hashCode()) % SIZE;
    if(buckets[index] == null) return null;
    for(MapEntry<K,V> iPair : buckets[index])
      if(iPair.getKey().equals(key))
        return iPair.getValue();
    return null;
  }
  public Set<Map.Entry<K,V>> entrySet() {
    Set<Map.Entry<K,V>> set= new HashSet<Map.Entry<K,V>>();
    for(LinkedList<MapEntry<K,V>> bucket : buckets) {
      if(bucket == null) continue;
      for(MapEntry<K,V> mpair : bucket)
        set.add(mpair);
    }
    return set;
  }
  public static void main(String[] args) {
    SimpleHashMap<String,String> m =
      new SimpleHashMap<String,String>();
    m.putAll(Countries.capitals(25));
    System.out.println(m);
    System.out.println(m.get("ERITREA"));
    System.out.println(m.entrySet());
  }
} /* Output:
```

```
{CAMEROON=Yaounde, CONGO=Brazzaville, CENTRAL AFRICAN
REPUBLIC=Bangui, GUINEA=Conakry, BOTSWANA=Gaberone,
BISSAU=Bissau, EGYPT=Cairo, ANGOLA=Luanda, BURKINA
FASO=Ouagadougou, ERITREA=Asmara, LESOTHO=Maseru,
THE GAMBIA=Banjul, KENYA=Nairobi, GABON=Libreville,
CHAD=N'djamena, CAPE VERDE=Praia, ALGERIA=Algiers,
COMOROS=Moroni, EQUATORIAL GUINEA=Malabo, BURUNDI=Bujumbura,
BENIN=Porto-Novo, COTE D'IVOIRE (IVORY COAST)=Yamoussoukro,
GHANA=Accra, DJIBOUTI=Dijibouti, ETHIOPIA=Addis Ababa}
Asmara
[CAMEROON=Yaounde, CONGO=Brazzaville, CENTRAL AFRICAN
REPUBLIC=Bangui, GUINEA=Conakry, BOTSWANA=Gaberone,
BISSAU=Bissau, EGYPT=Cairo, ANGOLA=Luanda, BURKINA
FASO=Ouagadougou, ERITREA=Asmara, LESOTHO=Maseru,
THE GAMBIA=Banjul, KENYA=Nairobi, GABON=Libreville,
CHAD=N'djamena, CAPE VERDE=Praia, ALGERIA=Algiers,
COMOROS=Moroni, EQUATORIAL GUINEA=Malabo, BURUNDI=Bujumbura,
BENIN=Porto-Novo, COTE D'IVOIRE (IVORY COAST)=Yamoussoukro,
GHANA=Accra, DJIBOUTI=Dijibouti, ETHIOPIA=Addis Ababa]
*///:~
```

Poiché le "posizioni" (*slot*) di una tabella hash spesso vengono chiamate *bucket,* all'array che rappresenta la tabella stessa è stato assegnato il nome **buckets**. Per favorire un'equa distribuzione degli oggetti nella tabella, il numero di bucket è generalmente rappresentato da un numero primo.[8]

Notate che si tratta di un array di **LinkedList**, che provvede automaticamente a risolvere le collisioni: ogni nuova voce viene aggiunta alla fine dell'elenco in un bucket particolare. Anche se Java non consente di creare un array di generici, potete sempre creare un *riferimento* a un array di questo tipo: in questo caso specifico si è ritenuto opportuno eseguire l'upcast a un array di generici, per evitare di dovere procedere a ulteriori upcasting in altri punti del codice.

8. È dimostrato che un numero primo non è il valore ideale per rappresentare le dimensioni dei bucket: dopo numerosi test, le versioni più recenti di Java hanno scelto di adottare per le implementazioni hash dimensioni pari a una potenza di 2 (*power of two*). Ancora oggi, la divisione e il calcolo del resto sono le operazioni più lente e complesse che un moderno processore possa svolgere. Servendosi invece di una tabella hash di dimensioni pari a una potenza di 2 in luogo della divisione, si può utilizzare la cosiddetta tecnica di *masking*, o maschera di bit. Poiché il metodo **get()** è di gran lunga l'operazione più frequente, il modulo (%) rappresenta una parte significativa dell'onere elaborativo, che viene annullato dall'approccio "*power of two*"; tenete presente, tuttavia, che il modulo coinvolge anche alcuni metodi **hashCode()**.

Ogni volta che viene chiamato il metodo **put()** viene chiamato anche il metodo **hashCode()** per la chiave corrente, e il risultato è convertito in un numero positivo. Per fare in modo che il numero risultante possa essere inserito nell'array **buckets**, che è di dimensioni fisse, ne viene calcolato il modulo con le dimensioni dell'array. Questo equivale a "normalizzare" il codice hash.

Se la posizione nell'array è **null**, significa che non ci sono elementi che possiedono un hash in quella posizione dell'array **buckets**, pertanto viene creata una nuova **LinkedList** per contenere l'oggetto che possiede l'hash da inserire in quella posizione. In ogni caso il processo normale prevede la ricerca di duplicati nella lista e, qualora ve ne siano, il salvataggio in **oldValue** del vecchio valore e la sua sostituzione con quello nuovo.

Il flag **found** tiene traccia del fatto che è stata trovata una vecchia coppia di chiave-valore, e, in caso negativo, la nuova coppia viene accodata alla lista.

Il metodo **get()** calcola l'indice nell'array **buckets** allo stesso modo di **put()**: questo è importante per garantire che vi ritroviate nello stesso punto. Se esiste una **LinkedList**, viene eseguita la ricerca di una corrispondenza.

Notate che questa implementazione non è ottimizzata in termini di prestazioni, ma il suo unico scopo è illustrare le operazioni possibili con una mappa di hash: per vedere un'implementazione ottimizzata, esaminate il codice sorgente di **java.util.HashMap**. Inoltre, per ragioni di praticità, **SimpleHashMap** utilizza lo stesso approccio a **entrySet()** di **SlowMap**, che essendo eccessivamente semplificato non è adatto per una **Map** di utilizzo quotidiano.

**Esercizio 19** (1) Ripetete l'Esercizio 13 servendovi di una **SimpleHashMap**.

**Esercizio 20** (3) Modificate **SimpleHashMap** per fare in modo che segnali le collisioni e testate questo meccanismo aggiungendo due volte lo stesso insieme di dati, verificando che vengano visualizzate le collisioni.

**Esercizio 21** (2) Modificate **SimpleHashMap** in modo che segnali il numero di "tentativi" necessari in caso di collisione: in pratica, quante chiamate a **next()** devono essere eseguite sull'**Iterator** che esamina le **LinkedList** in cerca di corrispondenze?

**Esercizio 22** (4) Implementate i metodi **clear()** e **remove()** per **SimpleHashMap**.

**Esercizio 23** (3) Implementate il resto dell'interfaccia **Map** per **SimpleHashMap**.

**Esercizio 24** (5) Sulla falsariga di **SimpleHashMap.java**, create e testate un **SimpleHashSet**.

**Esercizio 25** (6) Anziché utilizzare un **ListIterator** per ogni bucket, modificate **MapEntry** in modo che diventi una lista "a collegamento singolo" (*singly linked list*) che non fa ricorso a contenitori esterni, nella quale ogni **MapEntry** abbia un collegamento alla **MapEntry** successiva. Modificate il resto del codice in **SimpleHashMap.java** affinché funzioni correttamente con questa nuova tecnica.

## Sovrascrittura del metodo hashCode()

Ora che conoscete il funzionamento dell'hashing, la scrittura del vostro primo metodo **hashCode()** assumerà un maggiore significato.

In primo luogo, non avete controllo sulla creazione del valore reale utilizzato per indicizzare l'array di bucket. Questo dipende dalla capienza dell'oggetto **HashMap** che state utilizzando; il cambiamento di capienza dipende dal livello di riempimento del contenitore e dal valore del *coefficiente di carico* (*load factor*), di cui si parlerà in seguito. Pertanto, il valore prodotto dal vostro metodo **hashCode()** dovrà essere ulteriormente elaborato per creare l'indice del bucket; in **SimpleHashMap**, il calcolo è semplicemente un modulo della dimensione dell'array di bucket.

Il fattore più importante nella creazione di **hashCode()** è che, indipendentemente dal momento in cui **hashCode()** viene chiamato, produce sempre lo stesso valore per ogni particolare oggetto. Se avete un oggetto che produce un determinato valore di **hashCode()** in fase di inserimento (**put()**) in una **HashMap** e un valore diverso in fase di recupero (**get()**), non potrete recuperare gli oggetti. Di conseguenza, se il vostro **hashCode()** dipende dai dati modificabili dell'oggetto, il programmatore-utente dovrà essere informato che una modifica ai dati produrrà una chiave *differente* perché genera un **hashCode()** *differente*.

Inoltre, probabilmente *non* vorrete generare un **hashCode()** basato su una sola informazione dell'oggetto: tenete presente che il valore di **this**, in particolare, produce un **hashCode()** difettoso poiché non vi consente di creare una nuova chiave identica a quella utilizzata per inserire (**put()**), la coppia chiave-valore originale. Questo è lo stesso problema che si è verificato in **SpringDetector.java**, perché l'implementazione predefinita di **hashCode()** *si serve* dell'indirizzo dell'oggetto. Quindi abbiate cura di utilizzare informazioni contenute nell'oggetto che lo identifichino in modo espressivo.

Potete vedere un esempio nella classe **String**. Le **String** hanno una caratteristica particolare: se un programma contiene un numero elevato di oggetti **String** i quali, a loro volta, contengono identiche sequenze di caratteri, tali oggetti **String** punteranno tutti alla stessa area di memoria. È dunque sensa-

to che il valore **hashCode()** prodotto da due diverse **String "hello"** sia identico. Potete averne la prova nel programma seguente:

```
//: containers/StringHashCode.java

public class StringHashCode {
  public static void main(String[] args) {
    String[] hellos = "Hello Hello".split(" ");
    System.out.println(hellos[0].hashCode());
    System.out.println(hellos[1].hashCode());
  }
} /* Output:
69609650
69609650
*///:~
```

Il valore di **hashCode()** per **String** è chiaramente basato sul contenuto della **String**.

Di conseguenza, per essere efficace, un **hashCode()** deve essere veloce ed espressivo, ossia deve generare un valore basato sul contenuto dell'oggetto. Ricordate che questo valore non deve essere univoco: dovreste preoccuparvi più della velocità di esecuzione che dell'unicità. Tuttavia, tenete anche presente che tra **hashCode()** ed **equals()** l'identità dell'oggetto deve essere completamente definita.

Dato che **hashCode()** viene elaborato ulteriormente prima di produrre l'indice del bucket, l'intervallo di valori non è importante, perché serve soltanto per generare un valore **int**.

Esiste un altro fattore di cui si deve tenere conto: un buon metodo **hashCode()** dovrebbe generare una distribuzione uniforme di valori. Se i valori tendono a raggrupparsi, allora l'oggetto **HashMap** o **HashSet** sarà caricato più lentamente in alcune aree e non sarà così veloce come potrebbe esserlo invece con una funzione di hashing che genera una distribuzione casuale.

In *Effective Java Programming Language Guide* (Addison-Wesley, 2001), Joshua Bloch fornisce una serie di indicazioni per la generazione di valori **hashCode()** accettabili:

1. Memorizzate un valore costante diverso da zero, per esempio 17, in una variabile **int** di nome **result**.
2. Per ogni campo significativo **f** contenuto nel vostro oggetto, cioè per ogni campo considerato dal metodo **equals()**, calcolate un codice hash **int c**.

| *Tipo di campo* | *Calcolo* |
|---|---|
| **boolean** | **c = (f ? 0 : 1)** |
| **byte**, **char**, **short** o **int** | **c = (int)f** |
| **long** | **c = (int)(f ^ (f >>>32))** |
| **float** | **c = Float.floatToIntBits(f);** |
| **double** | **long l = Double.doubleToLongBits(f);** **c = (int)(l ^ (l >>> 32))** |
| **Object**, dove **equals()** chiama **equals()** per il campo corrente | **c = f.hashCode()** |
| **Array** | Le regole di calcolo elencate si applicano a ciascun elemento |

3. Combinate il codice o i codici hash calcolati: **result = 37 * result + c**.
4. Restituite **result**.
5. Verificate il valore **hashCode()** risultante e assicuratevi che istanze uguali abbiano valori di hash uguali.

L'esempio seguente riprende queste linee guida:

```
//: containers/CountedString.java
// Creazione di un buon codice hash con hashCode().
import java.util.*;
import static net.mindview.util.Print.*;

public class CountedString {
  private static List<String> created =
    new ArrayList<String>();
  private String s;
  private int id = 0;
  public CountedString(String str) {
    s = str;
    created.add(s);
    // id è il numero totale di istanze
    // di questa string usate da CountedString:
    for(String s2 : created)
      if(s2.equals(s))
```

```
      id++;
  }
  public String toString() {
    return "String: " + s + " id: " + id +
      " hashCode(): " + hashCode();
  }
  public int hashCode() {
    // L'approccio "semplicissimo":
    // return s.hashCode() * id;
    // L'approccio che tiene conto delle indicazioni di
    // Joshua Bloch:
    int result = 17;
    result = 37 * result + s.hashCode();
    result = 37 * result + id;
    return result;
  }
  public boolean equals(Object o) {
    return o instanceof CountedString &&
      s.equals(((CountedString)o).s) &&
      id == ((CountedString)o).id;
  }
  public static void main(String[] args) {
    Map<CountedString,Integer> map =
      new HashMap<CountedString,Integer>();
    CountedString[] cs = new CountedString[5];
    for(int i = 0; i < cs.length; i++) {
      cs[i] = new CountedString("hi");
      map.put(cs[i], i); // Autoboxing int -> Integer
    }
    print(map);
    for(CountedString cstring : cs) {
      print("Looking up " + cstring);
      print(map.get(cstring));
    }
  }
} /* Output:
```

```
{String: hi id: 4 hashCode(): 146450=3, String: hi id: 1
hashCode(): 146447=0, String: hi id: 3 hashCode():
146449=2, String: hi id: 5 hashCode(): 146451=4, String: hi
id: 2 hashCode(): 146448=1}
Looking up String: hi id: 1 hashCode(): 146447
0
Looking up String: hi id: 2 hashCode(): 146448
1
Looking up String: hi id: 3 hashCode(): 146449
2
Looking up String: hi id: 4 hashCode(): 146450
3
Looking up String: hi id: 5 hashCode(): 146451
4
*///:~
```

**CountedString** include una **String** e un **id** che rappresenta il numero di oggetti **CountedString** contenenti un valore **String** identico. Il conteggio viene eseguito nel costruttore iterando lo **static ArrayList** in cui sono memorizzate tutte le **String**.

Sia **hashCode()** sia **equals()** forniscono risultati basati su entrambi i campi (**id** e **String**); se infatti si basassero soltanto su **String** o su **id**, sarebbero presenti corrispondenze multiple per valori distinti.

Nel metodo **main()** vengono creati numerosi oggetti **CountedString** utilizzando lo stesso valore **String**, per evidenziare che i duplicati creano valori univoci a causa del diverso **id**. La **HashMap** viene visualizzata in modo che possiate valutarne il sistema interno di archiviazione; notate che non è rilevabile alcun ordinamento. Viene quindi ricercata ogni chiave, per dimostrare il funzionamento del meccanismo di ricerca.

Come secondo esempio, considerate la classe **Individual** utilizzata come classe di base per la libreria **typeinfo.pet**, definita nel Capitolo 2. In questo capitolo è stata utilizzata la classe **Individual**, posticipandone la definizione in modo che possiate comprenderne appieno l'implementazione:

```
//: typeinfo/pets/Individual.java
package typeinfo.pets;

public class Individual implements Comparable<Individual> {
```

```
private static long counter = 0;
private final long id = counter++;
private String name;
public Individual(String name) { this.name = name; }
// 'name' e' opzionale:
public Individual() {}
public String toString() {
  return getClass().getSimpleName() +
    (name == null ? "" : " " + name);
}
public long id() { return id; }
public boolean equals(Object o) {
  return o instanceof Individual &&
    id == ((Individual)o).id;
}
public int hashCode() {
  int result = 17;
  if(name != null)
    result = 37 * result + name.hashCode();
  result = 37 * result + (int)id;
  return result;
}
public int compareTo(Individual arg) {
  // Confronta per primi i nomi delle classi:
  String first = getClass().getSimpleName();
  String argFirst = arg.getClass().getSimpleName();
  int firstCompare = first.compareTo(argFirst);
  if(firstCompare != 0)
  return firstCompare;
  if(name != null && arg.name != null) {
    int secondCompare = name.compareTo(arg.name);
    if(secondCompare != 0)
      return secondCompare;
  }
  return (arg.id < id ? -1 : (arg.id == id ? 0 : 1));
}
```

```
} ///:~
```

Il metodo **compareTo()** dispone di una gerarchia di confronti, per produrre una sequenza che è ordinata innanzitutto per tipo effettivo, poi per nome (**name**, se presente) e infine per ordine di creazione. Il programma seguente dimostra il funzionamento di questi confronti:

```
//: containers/IndividualTest.java
import holding.MapOfList;
import typeinfo.pets.*;
import java.util.*;

public class IndividualTest {
  public static void main(String[] args) {
    Set<Individual> pets = new TreeSet<Individual>();
    for(List<? extends Pet> lp :
        MapOfList.petPeople.values())
      for(Pet p : lp)
        pets.add(p);
    System.out.println(pets);
  }
} /* Output:
[Cat Elsie May, Cat Pinkola, Cat Shackleton, Cat Stanford aka
Stinky el Negro, Cymric Molly, Dog Margrett, Mutt Spot, Pug
Louie aka Louis Snorkelstein Dupree, Rat Fizzy, Rat Freckly,
Rat Fuzzy]
*///:~
```

Poiché tutti questi animali hanno un nome, sono ordinati in primo luogo per tipo, quindi, all'interno dello stesso tipo, per nome.

La scrittura di metodi **hashCode()** e **equals()** adeguati in una nuova classe può essere difficile. Troverete alcune utility che vi aiuteranno in questa incombenza nel progetto "Jakarta Commons" di Apache (*http://jakarta.apache.org/commons/lang/*): questa iniziativa mette a disposizione molte altre librerie potenzialmente utili e rappresenta la risposta della comunità Java al progetto *www.boost.org* della comunità C++.

**Esercizio 26** (2) In **CountedString**, aggiungete un campo **char** che venga inizializzato nel costruttore e modificate i metodi **hashCode()** ed **equals()** per includere il valore di questo **char**.

**Esercizio 27** (3) Modificate il metodo **hashCode()** in **CountedString.java** rimuovendo la combinazione con **id**, e dimostrate che **CountedString** continua a funzionare come chiave. Quale problema presenta questo approccio?

**Esercizio 28** (4) Modificate il codice di **net/mindview/util/Tuple.java** per trasformarlo in una classe a utilizzo generico, aggiungendovi i metodi **hashCode()** ed **equals()** e implementando **Comparable** per ogni tipo di **Tuple**.

## Scegliere un'implementazione

A questo punto sapete che, benché esistano soltanto quattro tipi di contenitori fondamentali (**Map**, **List**, **Set** e **Queue**), ogni interfaccia può presentare più implementazioni. È quindi naturale che, dovendo servirvi delle funzionalità offerte da una determinata interfaccia, vi domandiate quale specifica implementazione utilizzare.

Ogni implementazione ha caratteristiche, punti di forza e debolezze che le sono peculiari. Per esempio, nel diagramma all'inizio di questo capitolo potete vedere che la "caratteristica" di **Hashtable**, **Vector** e **Stack** è che sono classi *legacy*, ancora presenti per garantire il funzionamento del vecchio codice e che è comunque preferibile evitare l'utilizzo nella creazione di nuovo codice.

I diversi tipi di **Queue** disponibili nella libreria Java si differenziano unicamente per il modo in cui accettano e restituiscono i valori: vedrete l'importanza di queste classi nel Volume 3, Capitolo 1.

La distinzione tra i vari tipi di contenitori spesso giunge a considerarne la "struttura portante", vale a dire le strutture dati che implementano fisicamente l'interfaccia desiderata. Per esempio, poiché **ArrayList** e **LinkedList** implementano entrambe l'interfaccia **List**, le operazioni *di base* di **List** sono le stesse. Tuttavia **ArrayList** è supportato da un array mentre **LinkedList** è implementato nel modo classico, mediante una lista "a collegamento doppio" (*doubly linked list*) nella quale i singoli oggetti contengono ciascuno i dati e i riferimenti agli elementi precedente e successivo. Per questo motivo, se dovete inserire e rimuovere molti elementi nella parte centrale di una lista, la scelta adatta cadrà su **LinkedList**: questa classe possiede anche altre funzionalità che sono parte della classe astratta **AbstractSequentialList**. Negli altri casi, in genere è preferibile un **ArrayList**.

Come ulteriore esempio, un **Set** può essere implementato come **TreeSet**, **HashSet** o **LinkedHashSet**.[9]

Ogni **Set** ha comportamenti diversi: **HashSet** è per un utilizzo generico e privilegia la velocità di ricerca, **LinkedHashSet** mantiene le coppie chiave-valore in ordine di inserimento e **TreeSet** è basato su **TreeMap** e progettato per creare un elenco costantemente ordinato. Abbiate cura di scegliere la variante opportuna in funzione del comportamento che meglio si adatta alla situazione.

Talvolta implementazioni differenti di un particolare contenitore condividono alcune operazioni, ma le prestazioni di tali operazioni potrebbero essere differenti; in tal caso dovrete valutare la possibile frequenza di utilizzo di una particolare funzionalità e la velocità che vi occorre. In questi casi, un modo per valutare le differenze tra le varie implementazioni dei contenitori consiste nel sottoporre questi ultimi a test prestazionali.

## *Struttura per test prestazionali*

Per impedire la duplicazione del codice e garantire test coerenti, l'autore ha scelto di inserire le funzionalità di base del processo di test in una struttura di massima, o *framework*. Il seguente codice crea una classe di base da cui viene creato un elenco di classi interne anonime, una per ogni test.

Ciascuna di queste classi interne viene chiamata come parte del processo di valutazione. Questo meccanismo vi consente di aggiungere o rimuovere facilmente alcune categorie di test.

Questo è un altro esempio del design pattern *Template Method*; in questo caso, tuttavia, anche se adottate il tipico approccio di questo modello (che consiste nel sovrascrivere il metodo **Test.test()** per ogni singolo test), il codice di base (che rimane invariato) si trova in una classe **Tester** separata.[10]

Il tipo di contenitore in test è il parametro generico **C**:

```
//: containers/Test.java
// Framework per l'esecuzione di test temporizzati
// sui contenitori.
```

9. **Set** può anche essere implementato come **EnumSet** e **CopyOnWriteArraySet**, che sono implementazioni particolari. Pur tenendo presente che possono esistere altre implementazioni specializzate delle varie interfacce di contenitori, in questo capitolo si è scelto di analizzare i casi più generali.

10. Krzysztof Sobolewski ha aiutato l'autore per i generici di questo esempio.

```
public abstract class Test<C> {
  String name;
  public Test(String name) { this.name = name; }
  // Sovrascrivete questo metodo per i diversi test.
  // Restituisce il numero di ripetizioni del test.
  abstract int test(C container, TestParam tp);
} ///:~
```

Ogni oggetto **Test** memorizza il nome del test. Quando chiamate il metodo **test()**, come primo argomento dovete passare il contenitore da esaminare e come secondo un "messaggero" detto anche "oggetto di trasferimento dati" (*DTO, Data Transfer Object*), ovvero un oggetto che contiene i parametri necessari per il test. I parametri includono **size**, che indica il numero di elementi presenti nel contenitore e **loops**, che imposta il numero di ripetizioni della prova: l'utilizzo di questi parametri nei singoli test è facoltativo.

Ogni contenitore eseguirà una serie di chiamate a **test()**, ciascuna con un diverso contenitore **TestParam**; in quest'ultimo sono presenti anche i metodi **static array()**, che semplificano la creazione di array di oggetti **TestParam**. La prima versione di **array()** accetta un elenco di argomenti variabile, contenente valori di **size** e **loops** alternati. La seconda versione accetta lo stesso tipo di lista, salvo che i valori vengono registrati sotto forma di **String**; in questo modo può essere utilizzata per gestire gli argomenti da riga di comando:

```
//: containers/TestParam.java
// Un oggetto "data transfer object."

public class TestParam {
  public final int size;
  public final int loops;
  public TestParam(int size, int loops) {
    this.size = size;
    this.loops = loops;
  }
  // Crea un array di TestParam da una sequenza vararg:
  public static TestParam[] array(int... values) {
    int size = values.length/2;
    TestParam[] result = new TestParam[size];
    int n = 0;
```

```
    for(int i = 0; i < size; i++)
      result[i] = new TestParam(values[n++], values[n++]);
    return result;
  }
  // Converte un array di String in un array TestParam:
  public static TestParam[] array(String[] values) {
    int[] vals = new int[values.length];
    for(int i = 0; i < vals.length; i++)
      vals[i] = Integer.decode(values[i]);
    return array(vals);
  }
} ///:~
```

Per servirvi di questo framework passate il contenitore da esaminare, insieme a una **List** di oggetti **Test**, a un metodo **Tester.run()**: questi sono metodi di comodo generici sovraccaricati, che riducono la quantità di codice necessario per il loro successivo utilizzo. Il metodo **Tester.run()** chiama il costruttore sovraccaricato opportuno, poi chiama **timedTest()**, che esegue ogni test contenuto nella lista di parametri associata al contenitore. Il metodo **timedTest()** ripete ogni test per ciascun oggetto **TestParam** presente in **paramList**. Dal momento che la **paramList** viene inizializzata dall'array statico **defaultParams**, potete modificare la **paramList** per tutti i test riassegnando **defaultParams**, oppure cambiare **paramList** per un solo test passando una **paramList** personalizzata per il test in questione:

```
//: containers/Tester.java
// Applica gli oggetti Test a liste di vari contenitori.
import java.util.*;

public class Tester<C> {
  public static int fieldWidth = 8;
  public static TestParam[] defaultParams= TestParam.array(
    10, 5000, 100, 5000, 1000, 5000, 10000, 500);
  // Sovrascrivete questo metodo per modificare
  // l'inizializzazione pre-test:
  protected C initialize(int size) { return container; }
  protected C container;
  private String headline = "";
```

```
private List<Test<C>> tests;
private static String stringField() {
  return "%" + fieldWidth + "s";
}
private static String numberField() {
  return "%" + fieldWidth + "d";
}
private static int sizeWidth = 5;
private static String sizeField = "%" + sizeWidth + "s";
private TestParam[] paramList = defaultParams;
public Tester(C container, List<Test<C>> tests) {
  this.container = container;
  this.tests = tests;
  if(container != null)
    headline = container.getClass().getSimpleName();
}
public Tester(C container, List<Test<C>> tests,
    TestParam[] paramList) {
  this(container, tests);
  this.paramList = paramList;
}
public void setHeadline(String newHeadline) {
  headline = newHeadline;
}
// Metodi di comodo generici:
public static <C> void run(C cntnr, List<Test<C>> tests){
  new Tester<C>(cntnr, tests).timedTest();
}
public static <C> void run(C cntnr,
    List<Test<C>> tests, TestParam[] paramList) {
  new Tester<C>(cntnr, tests, paramList).timedTest();
}
private void displayHeader() {
  // Calcola width ed integra il campo con '-':
  int width = fieldWidth * tests.size() + sizeWidth;
  int dashLength = width - headline.length() - 1;
```

```
    StringBuilder head = new StringBuilder(width);
    for(int i = 0; i < dashLength/2; i++)
      head.append('-');
    head.append(' ');
    head.append(headline);
    head.append(' ');
    for(int i = 0; i < dashLength/2; i++)
      head.append('-');
    System.out.println(head);
    // Visualizza le intestazioni di colonna:
    System.out.format(sizeField, "size");
    for(Test test : tests)
      System.out.format(stringField(), test.name);
    System.out.println();
  }
  // Esegue i test per il contenitore corrente:
  public void timedTest() {
    displayHeader();
    for(TestParam param : paramList) {
      System.out.format(sizeField, param.size);
      for(Test<C> test : tests) {
        C kontainer = initialize(param.size);
        long start = System.nanoTime();
        // Chiama il metodo sovrascritto:
        int reps = test.test(kontainer, param);
        long duration = System.nanoTime() - start;
        long timePerRep = duration / reps; // Nanoseconds
        System.out.format(numberField(), timePerRep);
      }
      System.out.println();
    }
  }
} ///:~
```

I metodi **stringField()** e **numberField()** creano stringhe di formattazione per la visualizzazione dei risultati; la larghezza standard di formattazione può es-

sere cambiata modificando il valore **static fieldWidth**. Il metodo **displayHeader()** formatta e visualizza le informazioni di intestazione per ciascun test.

Se dovete eseguire un'inizializzazione particolare, sovrascrivete il metodo **initialize()**, in modo da generare un oggetto contenitore inizializzato della dimensione appropriata; potete sia modificare il contenitore esistente sia crearne uno nuovo. Come vedete, nel metodo **test()** il risultato viene inserito in un riferimento locale chiamato **kontainer**; potete sostituire il membro **container** memorizzato con un contenitore inizializzato completamente diverso.

Il valore restituito da ogni metodo **Test.test()** deve essere il numero di operazioni eseguite da quel test; tale valore viene poi utilizzato per calcolare il numero di nanosecondi richiesti per ogni operazione. Tenete presente che **System.nanoTime()** produce solitamente valori con una granularità superiore a uno, che è comunque diversa a seconda del tipo di computer e di sistema operativo: questa differenza sarà causa di una certa imprecisione nei risultati.

I risultati possono variare da computer a computer, pertanto questi test sono progettati solo per il confronto tra prestazioni di contenitori differenti.

## Valutazione delle List

Di seguito è presentato un test prestazionale per le operazioni basilari di **List** nel quale, a titolo di confronto, sono mostrate anche le operazioni principali di **Queue**. Vengono create due liste di test separate per verificare ogni classe del contenitore; in questo caso specifico, le operazioni **Queue** si applicano unicamente a **LinkedList**.

```
//: containers/ListPerformance.java
// Dimostra le differenze prestazionali nelle List.
// {Args: 100 500} Da ridurre per abbreviare il test
import java.util.*;
import net.mindview.util.*;

public class ListPerformance {
  static Random rand = new Random();
  static int reps = 1000;
  static List<Test<List<Integer>>> tests =
    new ArrayList<Test<List<Integer>>>();
  static List<Test<LinkedList<Integer>>> qTests =
    new ArrayList<Test<LinkedList<Integer>>>();
  static {
```

```
tests.add(new Test<List<Integer>>("add") {
  int test(List<Integer> list, TestParam tp) {
    int loops = tp.loops;
    int listSize = tp.size;
    for(int i = 0; i < loops; i++) {
      list.clear();
      for(int j = 0; j < listSize; j++)
        list.add(j);
    }
    return loops * listSize;
  }
});
tests.add(new Test<List<Integer>>("get") {
  int test(List<Integer> list, TestParam tp) {
    int loops = tp.loops * reps;
    int listSize = list.size();
    for(int i = 0; i < loops; i++)
      list.get(rand.nextInt(listSize));
    return loops;
  }
});
tests.add(new Test<List<Integer>>("set") {
  int test(List<Integer> list, TestParam tp) {
    int loops = tp.loops * reps;
    int listSize = list.size();
    for(int i = 0; i < loops; i++)
      list.set(rand.nextInt(listSize), 47);
    return loops;
  }
});
tests.add(new Test<List<Integer>>("iteradd") {
  int test(List<Integer> list, TestParam tp) {
    final int LOOPS = 1000000;
    int half = list.size() / 2;
    ListIterator<Integer> it = list.listIterator(half);
    for(int i = 0; i < LOOPS; i++)
```

```
        it.add(47);
      return LOOPS;
    }
  });
  tests.add(new Test<List<Integer>>("insert") {
    int test(List<Integer> list, TestParam tp) {
      int loops = tp.loops;
      for(int i = 0; i < loops; i++)
        list.add(5, 47); // Minimizza l'onere per l'accesso
                         // casuale
      return loops;
    }
  });
  tests.add(new Test<List<Integer>>("remove") {
    int test(List<Integer> list, TestParam tp) {
      int loops = tp.loops;
      int size = tp.size;
      for(int i = 0; i < loops; i++) {
        list.clear();
        list.addAll(new CountingIntegerList(size));
        while(list.size() > 5)
          list.remove(5); // Minimizza l'onere per l'accesso
                          // casuale
      }
      return loops * size;
    }
  });
  // Verifica il comportamento della coda:
  qTests.add(new Test<LinkedList<Integer>>("addFirst") {
    int test(LinkedList<Integer> list, TestParam tp) {
      int loops = tp.loops;
      int size = tp.size;
      for(int i = 0; i < loops; i++) {
        list.clear();
        for(int j = 0; j < size; j++)
          list.addFirst(47);
```

```
      }
      return loops * size;
    }
  });
  qTests.add(new Test<LinkedList<Integer>>("addLast") {
    int test(LinkedList<Integer> list, TestParam tp) {
      int loops = tp.loops;
      int size = tp.size;
      for(int i = 0; i < loops; i++) {
        list.clear();
        for(int j = 0; j < size; j++)
          list.addLast(47);
      }
      return loops * size;
    }
  });
  qTests.add(
    new Test<LinkedList<Integer>>("rmFirst") {
      int test(LinkedList<Integer> list, TestParam tp) {
        int loops = tp.loops;
        int size = tp.size;
        for(int i = 0; i < loops; i++) {
          list.clear();
          list.addAll(new CountingIntegerList(size));
          while(list.size() > 0)
            list.removeFirst();
        }
        return loops * size;
      }
    });
  qTests.add(new Test<LinkedList<Integer>>("rmLast") {
    int test(LinkedList<Integer> list, TestParam tp) {
      int loops = tp.loops;
      int size = tp.size;
      for(int i = 0; i < loops; i++) {
        list.clear();
        list.addAll(new CountingIntegerList(size));
```

```
        while(list.size() > 0)
          list.removeLast();
      }
      return loops * size;
    }
  });
}
static class ListTester extends Tester<List<Integer>> {
  public ListTester(List<Integer> container,
      List<Test<List<Integer>>> tests) {
    super(container, tests);
  }
  // Popola la lista fino alla dimensione (size) appropriata
  // prima di ogni test:
  @Override protected List<Integer> initialize(int size){
    container.clear();
    container.addAll(new CountingIntegerList(size));
    return container;
  }
  // Metodo di comodo:
  public static void run(List<Integer> list,
      List<Test<List<Integer>>> tests) {
    new ListTester(list, tests).timedTest();
  }
}
public static void main(String[] args) {
  if(args.length > 0)
    Tester.defaultParams = TestParam.array(args);
  // Su un array si possono eseguire soltanto questi due
  // test:
  Tester<List<Integer>> arrayTest =
    new Tester<List<Integer>>(null, tests.subList(1, 3)){
      // Queste istruzioni vengono chiamate prima di ciascun
      // test; crea una lista di dimensioni fisse supportata
      // da un array:
      @Override protected
      List<Integer> initialize(int size) {
```

```
        Integer[] ia = Generated.array(Integer.class,
          new CountingGenerator.Integer(), size);
        return Arrays.asList(ia);
      }
    };
    arrayTest.setHeadline("Array as List");
    arrayTest.timedTest();
    Tester.defaultParams= TestParam.array(
      10, 5000, 100, 5000, 1000, 1000, 10000, 200);
    if(args.length > 0)
      Tester.defaultParams = TestParam.array(args);
    ListTester.run(new ArrayList<Integer>(), tests);
    ListTester.run(new LinkedList<Integer>(), tests);
    ListTester.run(new Vector<Integer>(), tests);
    Tester.fieldWidth = 12;
    Tester<LinkedList<Integer>> qTest =
      new Tester<LinkedList<Integer>>(
        new LinkedList<Integer>(), qTests);
    qTest.setHeadline("Queue tests");
    qTest.timedTest();
  }
} /* Output:
--- Array as List ---
 size     get     set
   10     130     183
  100     130     164
 1000     129     165
10000     129     165
--------------------- ArrayList ---------------------
 size     add     get     set iteradd  insert  remove
   10     121     139     191     435    3952     446
  100      72     141     191     247    3934     296
 1000      98     141     194     839    2202     923
10000     122     144     190    6880   14042    7333
--------------------- LinkedList ---------------------
 size     add     get     set iteradd  insert  remove
```

```
   10      182      164      198      658      366      262
  100      106      202      230      457      108      201
 1000      133     1289     1353      430      136      239
10000      172    13648    13187      435      255      239
---------------------- Vector ----------------------
 size      add      get      set  iteradd   insert   remove
   10      129      145      187      290     3635      253
  100       72      144      190      263     3691      292
 1000       99      145      193      846     2162      927
10000      108      145      186     6871    14730     7135
-------------------- Queue tests --------------------
 size     addFirst      addLast      rmFirst       rmLast
   10          199          163          251          253
  100           98           92          180          179
 1000           99           93          216          212
10000          111          109          262          384
*///:~
```

Dovete prestare la massima attenzione nella realizzazione di ogni test, per essere certi di produrre risultati significativi. Per esempio, il test "**add**" svuota la **List** e poi la popola fino alla dimensione specificata; la chiamata al metodo **clear()** fa quindi parte del test e può avere un impatto notevole sui tempi, specialmente per i test con un numero ridotto di valori inseriti. Anche se i risultati presentati sembrano ragionevoli, potreste pensare di riscrivere il framework del test in modo da prevedere una chiamata a un metodo preparatorio che, in questo caso, includerebbe la chiamata a **clear()** *fuori* del ciclo di conteggio dei tempi.

Ricordate che ogni test deve calcolare esattamente il numero di operazioni eseguite e che il metodo **test()** dovrà restituire questo valore, per consentirvi di calcolare correttamente i tempi.

I test "**get**" e "**set**" utilizzano entrambi il generatore di numeri casuali per eseguire gli accessi casuali alla **List**. L'output mostra che per una **List** supportata da un array e per un **ArrayList** questi accessi sono veloci e che la tempistica rimane costante a prescindere dalle dimensioni della lista, mentre per una **LinkedList** i tempi aumentano in modo notevole con liste di maggiori dimensioni. Chiaramente, questo dimostra che le **LinkedList** non sono la scelta appropriata qualora dobbiate eseguire numerosi accessi casuali.

Il test "**iteradd**" si serve di un iteratore per inserire i nuovi elementi nella parte centrale della lista. Per un **ArrayList** questa operazione è tanto più onerosa quanto più aumentano le dimensioni della lista, mentre per una **LinkedList** si tratta di un'operazione relativamente poco impegnativa, che si mantiene costante a prescindere dalle dimensioni. Questo comportamento è giustificato dal fatto che un **ArrayList** in fase di inserimento deve prima creare lo spazio, poi copiare tutti i riferimenti: operazioni tanto più gravose quanto più è grande l'**ArrayList**. Una **LinkedList**, invece, deve soltanto collegare un nuovo elemento senza modificare il resto della lista, pertanto l'onere dell'operazione è pressoché indipendente dalla dimensione della lista.

I test "**insert**" e "**remove**" utilizzano entrambi la posizione numero 5 come il punto di inserimento e di rimozione, invece dell'inizio o della fine della **List**. Una **LinkedList** tratta gli estremi delle **List** in modo particolare: in tal modo aumenta la velocità quando si utilizza una **LinkedList** come **Queue**. Tuttavia, se aggiungete o rimuovete elementi nella parte centrale della lista, dovrete prevedere un costo aggiuntivo per l'accesso casuale, che come si è visto varia a seconda delle diverse implementazioni di **List**. Eseguendo inserimenti e rimozioni nella posizione 5 tale onere dovrebbe essere trascurabile, quindi sarà valutato soltanto il carico di lavoro riferito alle operazioni di inserimento e rimozione; non si avvertirà tuttavia alcuna ottimizzazione specializzata per l'estremità di una **LinkedList**. L'output dimostra che il costo dell'inserimento e della rimozione in una **LinkedList** è relativamente limitato e non varia con la dimensione della lista; in un **ArrayList**, invece, gli inserimenti sono *molto* onerosi e tale costo aumenta con le dimensioni della lista.

Dai test eseguiti su **Queue** potete notare con quale rapidità una **LinkedList** inserisca e rimuova gli elementi dalle estremità della lista, il che è ottimale per il comportamento di **Queue**.

Di norma dovrete semplicemente chiamare **Tester.run()**, passando il contenitore e la lista **tests**. In questo caso, tuttavia, è necessario sovrascrivere il metodo **initialize()** in modo che la **List** venga svuotata e nuovamente riempita prima di ogni test: in caso contrario, il controllo della **List** sulle sue dimensioni si perderebbe durante i vari test. La classe **ListTester** eredita da **Tester** ed esegue questa inizializzazione servendosi di **CountingIntegerList**. Il metodo di comodo **run()** viene anch'esso sovrascritto.

È opportuno confrontare anche l'accesso agli array con quello ai contenitori, in modo particolare a fronte di **ArrayList**. Il metodo **main()** del primo test crea un oggetto **Test** particolare utilizzando una classe interna anonima. Ogni volta che viene chiamato, il metodo **initialize()** viene sovrascritto per creare un nuovo oggetto: l'oggetto **container** memorizzato viene ignorato e impostato a **null** per il costruttore **Tester** corrente. La creazione del nuovo oggetto avviene

per mezzo dei metodi **Generated.array()**, di cui si è parlato nel capitolo precedente, e **Arrays.asList()**. In questo caso possono essere eseguiti soltanto due test, poiché non è possibile inserire né rimuovere elementi quando si utilizza una **List** supportata da un array: di conseguenza, ci si serve del metodo **List.subList()** per selezionare i test opportuni a partire dall'elenco **tests**.

Per le operazioni **get()** e **set()** ad accesso casuale, una **List** supportata da un array è leggermente più veloce di un **ArrayList**; le stesse operazioni sono però molto più onerose per una **LinkedList**, poiché questa lista non è stata concepita per l'accesso casuale.

La classe **Vector** dovrebbe essere evitata, in quanto è presente nella libreria solo per supportare il codice di vecchie versioni Java: tenete presente che l'unica ragione per cui è in grado di funzionare con questo programma è perché è stata adattata, per ragioni di compatibilità, a essere una **List**.

L'approccio migliore in fase di valutazione delle prestazioni di una **List** consiste probabilmente nel privilegiare un **ArrayList** come scelta predefinita, passando a una **LinkedList** quando siano richieste funzionalità aggiuntive o qualora si rilevino problemi di prestazioni causati da un numero elevato di inserimenti o rimozioni di elementi nella parte centrale della lista. Se state lavorando con un gruppo di elementi a dimensione fissa potete utilizzare una **List** supportata da un array (prodotta da **Arrays.asList()**), oppure all'occorrenza un array vero e proprio.

La **CopyOnWriteArrayList** è un'implementazione particolare di **List** usata nella programmazione concorrente: se ne parlerà nel Volume 3, Capitolo 1.

> **Esercizio 29** (2) Modificate **ListPerformance.java** in modo che la **List** contenga oggetti **String**, anziché **Integer**. Servitevi di un **Generator** (Capitolo 4) per creare valori di test.

> **Esercizio 30** (3) Confrontate le prestazioni del metodo **Collections.sort()** tra un **ArrayList** e una **LinkedList**.

> **Esercizio 31** (5) Create un contenitore che incapsuli un array di **String** e che permetta di aggiungere e ottenere esclusivamente **String**, così da evitare operazioni di casting durante l'utilizzo. Se l'array interno non fosse abbastanza grande per l'inserimento successivo, il contenitore dovrà ridimensionarlo automaticamente. Nel metodo **main()** confrontate le prestazioni di questo contenitore con **ArrayList<String>**.

> **Esercizio 32** (2) Ripetete l'esercizio precedente creando un contenitore di **int** e confrontatene le prestazioni con un **ArrayList<Integer>**. Nel confronto prestazionale, includete l'operazione di incremento di ciascun oggetto nel contenitore.

**Esercizio 33** (5) Create un oggetto **FastTraversalLinkedList** che internamente utilizzi una **LinkedList** per la velocità delle operazioni di inserimento e rimozione, più un **ArrayList** per le operazioni **get()** trasversali, quindi sottoponetelo a test modificando **ListPerformance.java**.

## *Rischi impliciti nei microbenchmark*

Quando scrivete i cosiddetti *microbenchmark*, dovete prestare attenzione a non dare troppe cose per scontate e a limitare i vostri test in modo da cronometrare soltanto i tempi di ciò che vi interessa, per quanto possibile. Dovete anche assicurarvi che il test abbia una durata sufficiente per produrre dati significativi, e tenere ben presente che alcune tecnologie Java HotSpot ottimizzano i programmi soltanto se questi rimangono in esecuzione per un certo tempo. Quest'ultima considerazione è importante anche per programmi i cui tempi di esecuzione siano particolarmente brevi.

I test forniranno tempi di risposta diversi in base al computer e alla JVM utilizzati, pertanto dovreste eseguirli sul vostro computer per verificare la coerenza dei risultati con quelli indicati in questo volume. Inoltre è consigliabile che non vi concentriate sui valori assoluti dei risultati, confrontando invece i numeri riferiti a tipi di contenitori diversi.

In proposito, un *profiler* potrà eseguire un'analisi prestazionale meglio di quanto possiate fare voi stessi. Java è distribuito con un profiler: troverete maggiori informazioni nel supplemento *http://MindView.net/Books/BetterJava.* Java dispone di alcune utility di profiling e di analisi e rilevamento dei problemi (*troubleshooting*): per maggiori informazioni, potete consultare il documento disponibile all'indirizzo *http://java.sun.com/j2se/1.5/pdf/jdk50_ts_guide.pdf.* Sono disponibili anche profiler di terze parti, sia opensource sia proprietari.

Un esempio correlato interessa il metodo **Math.random()**. Questo metodo produce un valore da zero a uno, ma il valore "1" è compreso o escluso? Espresso in gergo matematico, è (0, 1), [0, 1], (0, 1] oppure [0, 1)? Le parentesi quadre indicano "incluso", mentre le parentesi tonde significano "non include". Il programma seguente *potrebbe* fornire la risposta:

```
//: containers/RandomBounds.java
// Math.random() restituisce 0.0 e 1.0?
// {RunByHand}
import static net.mindview.util.Print.*;

public class RandomBounds {
  static void usage() {
```

```
    print("Usage:");
    print("\tRandomBounds lower");
    print("\tRandomBounds upper");
    System.exit(1);
  }
  public static void main(String[] args) {
    if(args.length != 1) usage();
    if(args[0].equals("lower")) {
      while(Math.random() != 0.0)
        ; // Continua a provare
      print("Produced 0.0!");
    }
    else if(args[0].equals("upper")) {
      while(Math.random() != 1.0)
        ; // Continua a provare
      print("Produced 1.0!");
    }
    else
      usage();
  }
} ///:~
```

Per eseguire questo programma, dalla riga di comando digitate:

```
java RandomBounds lower
```

oppure

```
java RandomBounds upper
```

In entrambi i casi, sarete costretti a interrompere manualmente il programma con Ctrl + C, poiché sembra che **Math.random()** non produca mai i valori 0.0 né 1.0. Un esperimento di questo genere può risultare molto deludente: se considerate che ci sono circa 262 frazioni doppie differenti tra 0 e 1, la probabilità di ottenere sperimentalmente uno di questi valori supera la durata di vita del computer, e anche di chi attenta l'esperimento. Dalla documentazione risulta che 0.0 *è* incluso nell'output di **Math.random()**: in gergo matematico, è [0, 1). Pertanto, dovete analizzare i vostri test con attenzione e comprendere le limitazioni insite nel loro utilizzo.

### *Valutazione dei Set*

In base al comportamento che desiderate ottenere, potete scegliere fra **TreeSet**, **HashSet** o **LinkedHashSet**. Il programma di test seguente fornisce un confronto in termini di prestazioni fra queste tre varianti:

```
//: containers/SetPerformance.java
// Dimostra le differenze prestazionali nei Set.
// {Args: 100 5000} Da ridurre per abbreviare il test
import java.util.*;

public class SetPerformance {
  static List<Test<Set<Integer>>> tests =
    new ArrayList<Test<Set<Integer>>>();
  static {
    tests.add(new Test<Set<Integer>>("add") {
      int test(Set<Integer> set, TestParam tp) {
        int loops = tp.loops;
        int size = tp.size;
        for(int i = 0; i < loops; i++) {
          set.clear();
          for(int j = 0; j < size; j++)
            set.add(j);
        }
        return loops * size;
      }
    });
    tests.add(new Test<Set<Integer>>("contains") {
      int test(Set<Integer> set, TestParam tp) {
        int loops = tp.loops;
        int span = tp.size * 2;
        for(int i = 0; i < loops; i++)
          for(int j = 0; j < span; j++)
            set.contains(j);
        return loops * span;
      }
    });
```

```
    tests.add(new Test<Set<Integer>>("iterate") {
      int test(Set<Integer> set, TestParam tp) {
        int loops = tp.loops * 10;
        for(int i = 0; i < loops; i++) {
          Iterator<Integer> it = set.iterator();
          while(it.hasNext())
            it.next();
        }
        return loops * set.size();
      }
    });
  }
  public static void main(String[] args) {
    if(args.length > 0)
      Tester.defaultParams = TestParam.array(args);
    Tester.fieldWidth = 10;
    Tester.run(new TreeSet<Integer>(), tests);
    Tester.run(new HashSet<Integer>(), tests);
    Tester.run(new LinkedHashSet<Integer>(), tests);
  }
} /* Output:
------------- TreeSet -------------
 size       add  contains   iterate
   10       574       170        72
  100       464       269        50
 1000       653       418        54
10000      1416       543        56
------------- HashSet -------------
 size       add  contains   iterate
   10       244        91        91
  100       170        82        78
 1000       189       113        75
10000       440       138        74
---------- LinkedHashSet ----------
 size       add  contains   iterate
   10       263        75        57
```

```
  100       214       82      40
 1000       237      115      40
10000       925      132      42
*///:~
```

Le prestazioni di **HashSet** sono generalmente superiori a quelle di **TreeSet**, in particolare quando si aggiungono e si cercano gli elementi, le due operazioni principali. **TreeSet** ha la caratteristica di mantenere in ordine gli elementi, quindi è consigliabile quando vogliate gestire un **Set** ordinato. In considerazione della struttura interna necessaria a mantenere l'ordinamento *e* poiché l'iterazione è generalmente l'elaborazione più ricorrente, questa operazione di norma è più veloce in un **TreeSet** che in un **HashSet**.

Notate che le operazioni di inserimento in un **LinkedHashSet** sono più onerose rispetto a un **HashSet**, tenuto conto del maggiore lavoro sostenuto per supportare la lista collegata al contenitore con hash.

> **Esercizio 34** (1) Modificate **SetPerformance.java** in modo che **Set** contenga oggetti **String**, anziché **Integer**. Usate un **Generator** del Capitolo 4 per creare i valori di test.

## *Valutazione delle Map*

Il programma seguente fornisce un confronto tra le varie implementazioni di **Map**:

```
//: containers/MapPerformance.java
// Dimostra le differenze prestazionali nelle Map.
// {Args: 100 5000} Da ridurre per abbreviare il test
import java.util.*;

public class MapPerformance {
  static List<Test<Map<Integer,Integer>>> tests =
    new ArrayList<Test<Map<Integer,Integer>>>();
  static {
    tests.add(new Test<Map<Integer,Integer>>("put") {
      int test(Map<Integer,Integer> map, TestParam tp) {
        int loops = tp.loops;
        int size = tp.size;
        for(int i = 0; i < loops; i++) {
```

```
        map.clear();
        for(int j = 0; j < size; j++)
          map.put(j, j);
      }
      return loops * size;
    }
  });
  tests.add(new Test<Map<Integer,Integer>>("get") {
    int test(Map<Integer,Integer> map, TestParam tp) {
      int loops = tp.loops;
      int span = tp.size * 2;
      for(int i = 0; i < loops; i++)
        for(int j = 0; j < span; j++)
          map.get(j);
      return loops * span;
    }
  });
  tests.add(new Test<Map<Integer,Integer>>("iterate") {
    int test(Map<Integer,Integer> map, TestParam tp) {
      int loops = tp.loops * 10;
      for(int i = 0; i < loops; i ++) {
        Iterator it = map.entrySet().iterator();
        while(it.hasNext())
          it.next();
      }
      return loops * map.size();
    }
  });
}
public static void main(String[] args) {
  if(args.length > 0)
    Tester.defaultParams = TestParam.array(args);
  Tester.run(new TreeMap<Integer,Integer>(), tests);
  Tester.run(new HashMap<Integer,Integer>(), tests);
  Tester.run(new LinkedHashMap<Integer,Integer>(),tests);
  Tester.run(
```

```
      new IdentityHashMap<Integer,Integer>(), tests);
    Tester.run(new WeakHashMap<Integer,Integer>(), tests);
    Tester.run(new Hashtable<Integer,Integer>(), tests);
  }
} /* Output:
---------- TreeMap ----------
 size     put     get iterate
   10     648     166      74
  100     451     268      50
 1000     668     400      54
10000    1955     551      60
---------- HashMap ----------
 size     put     get iterate
   10     210      74      83
  100     341     105      90
 1000     224     112      76
10000     768     129      73
------- LinkedHashMap -------
 size     put     get iterate
   10     257      94      61
  100     203      83      44
 1000     259     114      44
10000    1422     138      39
------ IdentityHashMap ------
 size     put     get iterate
   10     308     208      82
  100     258     257      71
 1000     434     371      64
10000     622     402      58
-------- WeakHashMap --------
 size     put     get iterate
   10     354     164     110
  100     242     153      80
 1000     382     156     152
10000    1554     158     766
--------- Hashtable ---------
```

```
  size     put     get iterate
    10     219     127     125
   100     165     117      76
  1000     218     134      72
 10000     782     136      77
*///:~
```

Le operazioni di inserimento per tutte le implementazioni di **Map**, tranne **IdentityHashMap**, rallentano in modo notevole con l'aumentare delle dimensioni della **Map**; di solito, tuttavia, la ricerca è molto meno onerosa dell'inserimento (aspetto positivo, se si considera che la ricerca è sicuramente un'operazione più frequente).

Le prestazioni di **Hashtable** sono all'incirca equivalenti a quelle di **HashMap**: questo non dovrà sorprendervi, considerato che la **HashMap** è stata progettata per sostituire **Hashtable** e utilizza quindi lo stesso meccanismo di base per la consultazione e l'archiviazione, come vedrete in seguito.

La **TreeMap** è generalmente più lenta della **HashMap**. Come **TreeSet**, **TreeMap** è una tecnica per creare una lista ordinata. Il comportamento di un'alberatura di questo tipo è tale da essere sempre ordinato, pertanto non richiede alcun intervento. Dopo aver popolato una **TreeMap** potete chiamare il metodo **keySet()** per ottenere una vista delle chiavi come **Set**, e **toArray()** per produrre un array contenente tali chiavi; potrete poi servirvi del metodo **static Arrays.binarySearch()** per trovare rapidamente gli oggetti nell'array ordinato. Naturalmente tutto questo ha senso solo se ritenete il comportamento di una **HashMap** inaccettabile in termini di velocità, poiché di norma è **HashMap** la soluzione prevista per eseguire rapide ricerche delle chiavi. Inoltre, potete creare facilmente una **HashMap** da una **TreeMap** con un'unica creazione di oggetto o una chiamata del metodo **putAll()**. In conclusione, quando volete utilizzare una **Map** la vostra prima scelta dovrebbe cadere su **HashMap**: soltanto se vi occorre una **Map** costantemente ordinata dovreste scegliere una **TreeMap**.

**LinkedHashMap** tende a essere più lenta della **HashMap** in fase d'inserimento degli elementi, poiché gestisce una lista collegata per conservare l'ordine di inserimento, oltre alla struttura dati hash. Grazie alla presenza della lista collegata, l'iterazione è più veloce.

**IdentityHashMap** è caratterizzata da prestazioni diverse perché esegue i confronti ricorrendo all'operatore **==** invece che al metodo **equals()**. La **WeakHashMap** è descritta nel prosieguo del capitolo.

**Esercizio 35** (1) Modificate **MapPerformance.java** per includere i test di **SlowMap**.

**Esercizio 36** (5) Modificate **SlowMap** in modo che anziché due **ArrayList** gestisca un solo **ArrayList** di oggetti **MapEntry**, e verificate che la versione modificata funzioni correttamente. Quindi, servendovi di **MapPerformance.java**, verificate la velocità della vostra nuova **Map**; modificate poi il metodo **put()** in modo che esegua l'ordinamento con **sort()** dopo l'inserimento di ogni coppia chiave-valore e modificate **get()** affinché utilizzi **Collections.binarySearch()** per ricercare le chiavi. Confrontate le prestazioni delle due versioni.

**Esercizio 37** (2) Modificate **SimpleHashMap** per utilizzare un **ArrayList** anziché **LinkedList**, poi modificate **MapPerformance.java** per confrontare le prestazioni delle due implementazioni.

## Fattori che influenzano le prestazioni di HashMap

È possibile impostare manualmente i parametri di una **HashMap** per aumentarne le prestazioni nell'ambito di un'applicazione particolare. Prima di affrontare l'argomento delle prestazioni di una **HashMap**, è tuttavia necessario introdurre alcuni termini specifici che elenchiamo di seguito.

***Capienza***: il numero di bucket nella tabella.

***Capienza iniziale***: il numero di bucket al momento della creazione della tabella. **HashMap** e **HashSet** hanno costruttori che consentono di specificare la capienza iniziale.

***Dimensioni***: il numero di voci presenti in un determinato momento nella tabella.

***Coefficiente di carico***: rapporto tra dimensioni e capienza. Un coefficiente di carico di 0 equivale a una tabella vuota, 0,5 è una tabella piena a metà ecc. Una tabella poco carica avrà un numero limitato di collisioni e sarà quindi ottimale per gli inserimenti e le consultazioni, tuttavia rallenterà il processo di attraversamento con un iteratore. **HashMap** e **HashSet** hanno costruttori che consentono di specificare il coefficiente di carico: questo implica che al raggiungimento di questo valore il contenitore incrementerà automaticamente la sua capienza (il numero di bucket), all'incirca raddoppiandola, e gli oggetti attuali verranno ridistribuiti in una nuova serie di bucket, secondo una tecnica chiamata *rehashing*.

Il coefficiente di carico predefinito utilizzato da **HashMap** è 0,75: questo sembra essere un buon compromesso tra costi in termini di tempo e di spazio di archiviazione. Un coefficiente più elevato fa diminuire lo spazio richiesto

dalla tabella ma aumenta i tempi di consultazione, una considerazione importante dal momento che la consultazione è l'operazione più ricorrente, eseguita anche da **get()** e da **put()**.

Se ritenete che la vostra **HashMap** debba memorizzare molte voci, createla con una capienza iniziale sufficiente per evitare gli oneri derivanti dai successivi rehashing automatici.[11]

**Esercizio 38** (3) Consultate la documentazione JDK per la classe **HashMap**, quindi create una **HashMap**, popolatela di elementi e determinatene il coefficiente di carico. Verificate la velocità di consultazione di questa mappa, poi cercate di aumentare le prestazioni generando una nuova **HashMap** con una capienza iniziale maggiore e copiando la vecchia mappa nella nuova. Infine eseguite ancora i test sulla nuova mappa.

**Esercizio 39** (6) Aggiungete un metodo **private rehash()** a **SimpleHashMap** che venga chiamato quando il coefficiente di carico eccede il valore 0,75. In occasione del rehashing raddoppiate il numero di bucket, poi cercate il primo numero primo maggiore di quello utilizzato per determinare il nuovo numero di bucket.

## Utility

Esistono numerosi programmi di utilità specifici per i contenitori, espressi come metodi **static** all'interno della classe **java.util.Collections**; ne avete visti alcuni, quale **addAll()**, **reverseOrder()** e **binarySearch()**. La tabella di pagina seguente elenca le utility rimanenti; tenete presente che i programmi **synchronized** e **unmodifiable** saranno illustrati nei paragrafi che seguono. In questa tabella, quando opportuno, è segnalato l'utilizzo dei generici.

11. In un messaggio riservato all'autore, l'ex progettista di Sun Joshua Bloch ha scritto: "ritengo sia stato un errore permettere di definire i dettagli implementativi (quali le dimensioni della tabella hash e il coefficiente di carico) nelle nostre API. Forse sarebbe opportuno che il client fornisse le dimensioni massime previste per una collezione, e che Java si servisse di tali valori. È facile che i client risultino più dannosi che utili, scegliendo autonomamente i valori per questi parametri. Come esempio estremo, considerate il campo **capacityIncrement** di **Vector**: nessuno dovrebbe mai impostarlo, e noi stessi non avremmo dovuto metterli a disposizione del pubblico. Se viene impostato a qualsiasi valore diverso da zero, il costo asintotico di una sequenza di accodamenti va da lineare a quadratico: in altri termini, distrugge le vostre prestazioni. Con il tempo, stiamo progressivamente 'rinsavendo' su questo genere di cose. Se osservate **IdentityHashMap** vedrete che non ha parametri di sintonia a basso livello".

| | |
|---|---|
| **checkedCollection(Collection<T>, Class<T> type)**<br>**checkedList(List<T>, Class<T> type)**<br>**checkedMap(Map<K,V>, Class<K> keyType, Class<V> valueType)**<br>**checkedSet(Set<T>, Class<T> type)**<br>**checkedSortedMap(SortedMap<K, V>, Class<K> keyType, Class<V> valueType)**<br>**checkedSortedSet(SortedSet<T>, Class<T> type)** | Restituisce una "vista con sicurezza dinamica dei tipi" (*dynamically type-safe view*) di una **Collection**, o di uno specifico sottotipo di **Collection**. Tali metodi dovrebbero essere utilizzati quando non è possibile servirsi della versione a controllo statico dei tipi. Questi metodi sono stati illustrati nel paragrafo "Sicurezza di tipo dinamica" del Capitolo 3. |
| **max(Collection)**<br>**min(Collection)** | Restituisce l'elemento massimo o minimo presente nella collection passata come argomento, utilizzando il metodo naturale di confronto degli oggetti disponibile in **Collection**. |
| **max(Collection, Comparator)**<br>**min(Collection, Comparator)** | Restituisce l'elemento massimo o minimo presente in **Collection**, servendosi di **Comparator**. |
| **indexOfSubList(List source, List target)** | Restituisce l'indice iniziale della *prima* occorrenza di **target** all'interno di **source**, o **-1** se non esiste alcuna occorrenza. |
| **lastIndexOfSubList(List source, List target)** | Restituisce l'indice iniziale dell'*ultima* occorrenza di **target** all'interno di **source**, o **-1** se non esiste alcuna occorrenza. |
| **replaceAll(List<T>, T oldVal, T newVal)** | Sostituisce tutte le occorrenze di **oldVal** con **newVal**. |
| **reverse(List)** | Inverte l'ordine degli elementi nella lista specificata. |
| **reverseOrder()**<br>**reverseOrder(Comparator<T>)** | Restituisce un **Comparator** che inverte l'ordinamento naturale di una collezione di oggetti che implementano **Comparable<T>**. La seconda forma inverte l'ordine del **Comparator** fornito. |
| **rotate(List, int distance)** | Sposta tutti gli elementi in avanti a partire da **distance**, ricollocando all'inizio gli elementi che vanno oltre il termine della lista. |

| | |
|---|---|
| **shuffle(List)**<br>**shuffle(List, Random)** | Permuta a caso la lista specificata: la prima forma fornisce la propria sorgente di randomizzazione; la seconda accetta quella fornita dal programmatore. |
| **sort(List<T>)**<br>**sort(List<T>, Comparator<? super T> c)** | Ordina **List<T>** secondo il suo criterio naturale di ordinamento. La seconda forma consente di fornire un **Comparator** per l'ordinamento. |
| **copy(List<? super T> dest, List<? extends T> src)** | Copia gli elementi da **src** a **dest**. |
| **swap(List, int i, int j)** | Scambia gli elementi nelle posizioni **i** e **j** della **List**. |
| **fill(List<? super T>, T x)** | Sostituisce tutti gli elementi della lista con **x**. |
| **nCopies(int n, T x)** | Restituisce una **List<T>**, non modificabile, di dimensioni pari a **n**, nella quale tutti i riferimenti puntano a **x**. |
| **disjoint(Collection, Collection)** | Restituisce **true** se le due collezioni non hanno elementi in comune. |
| **frequency(Collection, Object x)** | Restituisce il numero di elementi nella **Collection** che sono uguali a **x**. |
| **emptyList()**<br>**emptyMap()**<br>**emptySet()** | Restituisce una **List**, una **Map** o un **Set** vuoti e non modificabili. Questi sono generici, in modo che la **Collection** risultante sarà parametrizzabile al tipo voluto. |
| **singleton(T x)**<br>**singletonList(T x)**<br>**singletonMap(K key, V value)** | Restituisce un **Set<T>**, una **List<T>** o una **Map<K, V>** non modificabili, contenenti un'unica voce basata sull'argomento o gli argomenti forniti. |
| **list(Enumeration<T> e)** | Restituisce un **ArrayList<T>** contenente gli elementi nell'ordine in cui vengono restituiti secondo i criteri previsti da **Enumeration**, il predecessore di **Iterator**. Questo metodo è utilizzato nelle conversioni di vecchio codice legacy. |
| **enumeration(Collection<T>)** | Restituisce una **Enumeration<T>** per l'argomento fornito. |

Notate che i metodi **min()** e **max()** funzionano solo con gli oggetti **Collection**, non con le **List**: pertanto non dovrete preoccuparvi se la **Collection** è ordinata oppure no: ricorderete, infatti, che prima di eseguire una **binarySearch()** è necessario sottoporre a **sort()** una **List** o un array.

Questo esempio mostra l'utilizzo di base della maggior parte delle utility descritte nella tabella precedente:

```
//: containers/Utilities.java
// Semplice dimostrazione delle utility per le Collection.
import java.util.*;
import static net.mindview.util.Print.*;

public class Utilities {
  static List<String> list = Arrays.asList(
    "one Two three Four five six one".split(" "));
  public static void main(String[] args) {
    print(list);
    print("'list' disjoint (Four)?: " +
      Collections.disjoint(list,
        Collections.singletonList("Four")));
    print("max: " + Collections.max(list));
    print("min: " + Collections.min(list));
    print("max w/ comparator: " + Collections.max(list,
      String.CASE_INSENSITIVE_ORDER));
    print("min w/ comparator: " + Collections.min(list,
      String.CASE_INSENSITIVE_ORDER));
    List<String> sublist =
      Arrays.asList("Four five six".split(" "));
    print("indexOfSubList: " +
      Collections.indexOfSubList(list, sublist));
    print("lastIndexOfSubList: " +
      Collections.lastIndexOfSubList(list, sublist));
    Collections.replaceAll(list, "one", "Yo");
    print("replaceAll: " + list);
    Collections.reverse(list);
    print("reverse: " + list);
    Collections.rotate(list, 3);
```

```
    print("rotate: " + list);
    List<String> source =
      Arrays.asList("in the matrix".split(" "));
    Collections.copy(list, source);
    print("copy: " + list);
    Collections.swap(list, 0, list.size() - 1);
    print("swap: " + list);
    Collections.shuffle(list, new Random(47));
    print("shuffled: " + list);
    Collections.fill(list, "pop");
    print("fill: " + list);
    print("frequency of 'pop': " +
      Collections.frequency(list, "pop"));
    List<String> dups = Collections.nCopies(3, "snap");
    print("dups: " + dups);
    print("'list' disjoint 'dups'?: " +
      Collections.disjoint(list, dups));
    // Come ottenere un'Enumeration legacy:
    Enumeration<String> e = Collections.enumeration(dups);
    Vector<String> v = new Vector<String>();
    while(e.hasMoreElements())
      v.addElement(e.nextElement());
    // Conversione di una classe Vector legacy
    // in una List, tramite un'Enumeration:
    ArrayList<String> arrayList =
      Collections.list(v.elements());
    print("arrayList: " + arrayList);
  }
} /* Output:
[one, Two, three, Four, five, six, one]
'list' disjoint (Four)?: false
max: three
min: Four
max w/ comparator: Two
min w/ comparator: five
indexOfSubList: 3
```

```
lastIndexOfSubList: 3
replaceAll: [Yo, Two, three, Four, five, six, Yo]
reverse: [Yo, six, five, Four, three, Two, Yo]
rotate: [three, Two, Yo, Yo, six, five, Four]
copy: [in, the, matrix, Yo, six, five, Four]
swap: [Four, the, matrix, Yo, six, five, in]
shuffled: [six, matrix, the, Four, Yo, five, in]
fill: [pop, pop, pop, pop, pop, pop, pop]
frequency of 'pop': 7
dups: [snap, snap, snap]
'list' disjoint 'dups'?: true
arrayList: [snap, snap, snap]
*///:~
```

L'output evidenzia il comportamento dei singoli metodi di utilità. Notate la differenza nei risultati dei metodi **min()** e **max()** con il **Comparator String.CASE_INSENSITIVE_ORDER**, a causa delle lettere maiuscole e minuscole.

## *Ordinamento e ricerche nelle List*

I metodi di utilità per eseguire l'ordinamento e la ricerca nelle **List** hanno gli stessi nomi e segnature di quelli per gli array di oggetti, tuttavia sono metodi **static** di **Collections** anziché di **Arrays**. Considerate questo esempio, che utilizza gli stessi dati della **list** di **Utilities.java**:

```
//: containers/ListSortSearch.java
// Ordinamento e ricerca in List con le utility
// per le Collection.
import java.util.*;
import static net.mindview.util.Print.*;

public class ListSortSearch {
  public static void main(String[] args) {
    List<String> list =
      new ArrayList<String>(Utilities.list);
    list.addAll(Utilities.list);
    print(list);
```

```
    Collections.shuffle(list, new Random(47));
    print("(Shuffle(: " + list);
    // Usa ListIterator per eliminare gli ultimi elementi:
    ListIterator<String> it = list.listIterator(10);
    while(it.hasNext()) {
      it.next();
      it.remove();
    }
    print("Trimmed: " + list);
    Collections.sort(list);
    print("Sorted: " + list);
    String key = list.get(7);
    int index = Collections.binarySearch(list, key);
    print("Location of " + key + " is " + index +
      ", list.get(" + index + ") = " + list.get(index));
    Collections.sort(list, String.CASE_INSENSITIVE_ORDER);
    print("Case-insensitive sorted: " + list);
    key = list.get(7);
    index = Collections.binarySearch(list, key,
      String.CASE_INSENSITIVE_ORDER);
    print("Location of " + key + " is " + index +
      ", list.get(" + index + ") = " + list.get(index));
  }
} /* Output:
[one, Two, three, Four, five, six, one, one, Two, three,
Four, five, six, one]
Shuffled: [Four, five, one, one, Two, six, six, three,
three, five, Four, Two, one, one]
Trimmed: [Four, five, one, one, Two, six, six, three,
three, five]
Sorted: [Four, Two, five, five, one, one, six, six, three,
three]
Location of six is 7, list.get(7) = six
Case-insensitive sorted: [five, five, Four, one, one, six,
six, three, three, Two]
Location of three is 7, list.get(7) = three
*///:~
```

Esattamente come avviene quando si ricercano e ordinano gli array, se eseguite l'ordinamento utilizzando un **Comparator** dovrete anche eseguire **binarySearch()** con lo stesso **Comparator**.

Questo programma dimostra inoltre il metodo **shuffle()** in **Collections**, che rende casuale l'ordine di una **List**: un **ListIterator** viene creato in una determinata posizione della lista mescolata ed è poi utilizzato per rimuovere gli elementi da quella posizione fino alla fine della **List**.

> **Esercizio 40** (5) Create una classe contenente due oggetti **String** e rendetela confrontabile (**Comparable**) in modo che il confronto tenga conto soltanto della prima **String**. Popolate un array e un **ArrayList** di oggetti della vostra classe, per mezzo del generatore **RandomGenerator**, e dimostrate che l'ordinamento è corretto. Ora create un **Comparator** che tenga conto unicamente della seconda **String** e dimostrate che l'ordinamento è corretto. Infine, eseguite una ricerca binaria servendovi del vostro **Comparator**.
>
> **Esercizio 41** (3) Modificate la classe dell'esercizio precedente in modo che funzioni con un **HashSet** e come chiave di **HashMap**.
>
> **Esercizio 42** (2) Modificate l'Esercizio 40 per utilizzare un ordinamento alfabetico.

## *Rendere immutabili una Collection o una Map*

Spesso è utile creare una versione di sola lettura di una **Collection** o una **Map**. La classe **Collections** offre questa funzionalità, consentendovi di passare il contenitore originale a un metodo che restituirà una versione di sola lettura del contenitore.

Esistono alcune varianti di questo metodo per le **Collection** (se non potete gestire una **Collection** come tipo più specifico), le **List**, i **Set** e le **Map**. Il codice seguente mostra il modo corretto per rendere di sola lettura ciascuno di questi oggetti:

```
//: containers/ReadOnly.java
// Utilizzo dei metodi di Collections.unmodifiable.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class ReadOnly {
  static Collection<String> data =
    new ArrayList<String>(Countries.names(6));
```

```
  public static void main(String[] args) {
    Collection<String> c =
      Collections.unmodifiableCollection(
        new ArrayList<String>(data));
    print(c); // OK in lettura
    //! c.add("one"); // Impossibile da modificare

    List<String> a = Collections.unmodifiableList(
        new ArrayList<String>(data));
    ListIterator<String> lit = a.listIterator();
    print(lit.next()); // OK in lettura
    //! lit.add("one"); // Impossibile da modificare

    Set<String> s = Collections.unmodifiableSet(
      new HashSet<String>(data));
    print(s); // OK in lettura
    //! s.add("one"); // Impossibile da modificare

    // Per un SortedSet:
    Set<String> ss = Collections.unmodifiableSortedSet(
      new TreeSet<String>(data));

    Map<String,String> m = Collections.unmodifiableMap(
      new HashMap<String,String>(Countries.capitals(6)));
    print(m); // OK in lettura
    //! m.put("Ralph", "Howdy!");

    // Per una SortedMap:
    Map<String,String> sm =
      Collections.unmodifiableSortedMap(
        new TreeMap<String,String>(Countries.capitals(6)));
  }
} /* Output:
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI]
ALGERIA
[BURKINA FASO, BOTSWANA, BENIN, ANGOLA, ALGERIA, BURUNDI]
```

```
{BURKINA FASO=Ouagadougou, BOTSWANA=Gaberone, BENIN=Porto-
Novo, ANGOLA=Luanda, ALGERIA=Algiers, BURUNDI=Bujumbura}
ALGERIA=Algiers}
*///:~
```

La chiamata di un metodo "unmodifiable" per un tipo particolare non attiva automaticamente il controllo in fase di compilazione, ma una volta che la trasformazione è avvenuta tutte le chiamate ai metodi che modificano il contenuto di un determinato contenitore produrranno una **UnsupportedOperationException**.

In qualsiasi caso, dovrete popolare il contenitore con dati significativi *prima* di renderlo di sola lettura. Una volta che è caricato, il modo migliore per renderlo immutabile è sostituire il riferimento attuale con quello prodotto dalla chiamata "unmodifiable". In questo modo non correrete il rischio di modificare inavvertitamente i contenuti dopo averlo reso "non modificabile". D'altra parte, questa utility consente anche di mantenere un contenitore modificabile come **private** all'interno di una classe e di restituire un riferimento di sola lettura per quel contenitore da una chiamata di metodo: potrete quindi eseguire modifiche dall'interno della classe, ma dall'esterno il contenitore continuerà a essere di sola lettura.

## *Sincronizzare una Collection o una Map*

La parola chiave **synchronized** è un elemento fondamentale del *multithreading*, un argomento complesso che vi sarà illustrato nel Volume 3, Capitolo 1. A questo punto è comunque opportuno segnalare che la classe **Collections** fornisce alcuni metodi per sincronizzare automaticamente un intero contenitore. La sintassi è analoga a quella dei metodi "unmodifiable":

```
//: containers/Synchronization.java
// Utilizzo dei metodi di Collections.synchronized.
import java.util.*;

public class Synchronization {
  public static void main(String[] args) {
    Collection<String> c =
      Collections.synchronizedCollection(
        new ArrayList<String>());
    List<String> list = Collections.synchronizedList(
      new ArrayList<String>());
```

```
    Set<String> s = Collections.synchronizedSet(
      new HashSet<String>());
    Set<String> ss = Collections.synchronizedSortedSet(
      new TreeSet<String>());
    Map<String,String> m = Collections.synchronizedMap(
      new HashMap<String,String>());
    Map<String,String> sm =
      Collections.synchronizedSortedMap(
        new TreeMap<String,String>());
  }
} ///:~
```

È preferibile passare immediatamente il nuovo contenitore al metodo "sincronizzato" adatto, come indicato sopra: in questo modo eviterete di rendere inavvertitamente pubblica una versione non sincronizzata.

## Meccanismo di fail-fast

I contenitori Java possiedono anche un meccanismo per impedire a più processi di modificare il contenuto di un contenitore. Il problema nasce se vi trovate nel mezzo di un'iterazione di un contenitore e altri processi si inseriscono cercando di aggiungere, rimuovere o modificare un oggetto nel contenitore. Forse nella vostra iterazione avete già superato quell'elemento nel contenitore, forse dovete ancora raggiungerlo, o magari la dimensione del contenitore si riduce in seguito alla chiamata del metodo **size()**; comunque sia, in questo caso si prospettano scenari disastrosi. Fortunatamente la libreria di contenitori Java utilizza un meccanismo *fail-fast* (che letteralmente significa "a interruzione rapida"), che controlla tutti i cambiamenti al contenitore tranne quelli di cui è responsabile il vostro processo: se rileva che altri processi stanno modificando il contenitore, genera immediatamente un'eccezione **ConcurrentModificationException**. È questo comportamento che rappresenta l'aspetto *fail-fast*: non perde tempo a rilevare il problema utilizzando un algoritmo complesso. È facilissimo vedere il meccanismo fail-fast in azione: basta creare un iteratore e poi cercare di aggiungere qualcosa alla collezione cui sta puntando l'iteratore, come nell'esempio seguente:

```
//: containers/FailFast.java
// Dimostrazione del comportamento "fail-fast".
import java.util.*;
```

```
public class FailFast {
  public static void main(String[] args) {
    Collection<String> c = new ArrayList<String>();
    Iterator<String> it = c.iterator();
    c.add("An object");
    try {
      String s = it.next();
    } catch(ConcurrentModificationException e) {
      System.out.println(e);
    }
  }
} /* Output:
java.util.ConcurrentModificationException
*///:~
```

L'eccezione si produce perché qualcosa viene inserito nel contenitore *dopo* che l'iterator è stato acquisito dal contenitore. La possibilità che due parti diverse del programma possano modificare lo stesso contenitore genera una condizione di incertezza, pertanto l'eccezione vi informa che dovreste modificare il codice. In questo caso, sarà sufficiente acquisire l'iteratore *dopo* aver aggiunto tutti gli elementi al contenitore.

Le classi **ConcurrentHashMap**, **CopyOnWriteArrayList** e **CopyOnWriteArraySet** utilizzano tecniche che evitano gli errori **ConcurrentModificationException**.

## Conservare i riferimenti

La libreria **java.lang.ref** contiene un insieme di classi che offrono una maggiore flessibilità nella garbage collection, particolarmente utili quando utilizzate oggetti di grandi dimensioni che potrebbero esaurire la memoria disponibile. Sono disponibili tre classi ereditate da quella astratta **Reference**: **SoftReference**, **WeakReference** e **PhantomReference**, ciascuna delle quali fornisce un diverso livello di indirezione per il garbage collector, se l'oggetto in questione è raggiungibile soltanto tramite uno di questi oggetti **Reference**.

Se un oggetto è *raggiungibile*, significa che può essere localizzato in qualche punto del vostro programma. Questo potrebbe implicare la presenza di un normale riferimento sullo stack, che punta direttamente all'oggetto; in realtà, potrebbe anche trattarsi di un riferimento a un oggetto che possiede un ulteriore riferimento all'oggetto in questione: possono esserci anche molti colle-

gamenti intermedi. Se un oggetto *è raggiungibile* il garbage collector non può rilasciarlo, in quanto è ancora utilizzato dal vostro programma; in caso contrario, se un oggetto *non è raggiungibile* il programma non può servirsene in alcun modo: il garbage collector potrà quindi eliminarlo in assoluta sicurezza.

Quando desiderate continuare a mantenere un riferimento a questo oggetto perché volete che rimanga possibile raggiungerlo, potete servirvi degli oggetti **Reference**; tuttavia, vorrete anche consentire al garbage collector di rilasciare la memoria occupata da questo oggetto. La tecnica di conservazione dei riferimenti vi offre quindi la possibilità di utilizzare l'oggetto: se l'esaurimento della memoria è imminente, consente di rilasciare l'oggetto.

Questo meccanismo richiede che un oggetto **Reference** agisca come "mediatore" (*proxy*) tra voi e il normale riferimento. Non devono inoltre esistere "normali" riferimenti all'oggetto che non siano stati spostati in oggetti **Reference**: infatti, qualora il garbage collector rilevi che un oggetto è raggiungibile tramite un normale riferimento, non lo rilascerà.

Ciascuno degli oggetti **SoftReference**, **WeakReference** e **PhantomReference** è "più debole" del precedente e corrisponde a un differente livello di raggiungibilità. I riferimenti *soft* sono impiegati per la creazione di cache *memory-sensitive*, ovvero "sensibili" alla quantità di memoria occupata. I riferimenti *weak* sono utilizzati per implementare le cosiddette *canonicalizing mapping* (letteralmente "mappature canoniche"), nelle quali alcune istanze di oggetti vengono utilizzate contemporaneamente in più punti del programma per risparmiare memoria: ciò non impedisce a queste chiavi o valori di essere eliminati dal garbage collector. I riferimenti *phantom* sono utilizzati nelle operazioni di cleanup cosiddette *pre-mortem*, che richiedono un meccanismo più flessibile di quello reso possibile dal sistema di finalizzazione di Java.

Con **SoftReference** e **WeakReference** potete inserire i riferimenti in una **ReferenceQueue**, il dispositivo adottato per le operazioni di cleanup pre-mortem; una **PhantomReference**, invece, può essere costruita esclusivamente su una **ReferenceQueue**. Considerate questa semplice dimostrazione:

```
//: containers/References.java
// Dimostrazione degli oggetti Reference
import java.lang.ref.*;
import java.util.*;

class VeryBig {
  private static final int SIZE = 10000;
  private long[] la = new long[SIZE];
```

```
  private String ident;
  public VeryBig(String id) { ident = id; }
  public String toString() { return ident; }
  protected void finalize() {
    System.out.println("Finalizing " + ident);
  }
}

public class References {
  private static ReferenceQueue<VeryBig> rq =
    new ReferenceQueue<VeryBig>();
  public static void checkQueue() {
    Reference<? extends VeryBig> inq = rq.poll();
    if(inq != null)
      System.out.println("In queue: " + inq.get());
  }
  public static void main(String[] args) {
    int size = 10;
    // Oppure scegliete le dimensioni dalla riga di comando:
    if(args.length > 0)
      size = new Integer(args[0]);
    LinkedList<SoftReference<VeryBig>> sa =
      new LinkedList<SoftReference<VeryBig>>();
    for(int i = 0; i < size; i++) {
      sa.add(new SoftReference<VeryBig>(
        new VeryBig("Soft " + i), rq));
      System.out.println("Just created: " + sa.getLast());
      checkQueue();
    }
    LinkedList<WeakReference<VeryBig>> wa =
      new LinkedList<WeakReference<VeryBig>>();
    for(int i = 0; i < size; i++) {
      wa.add(new WeakReference<VeryBig>(
        new VeryBig("Weak " + i), rq));
      System.out.println("Just created: " + wa.getLast());
      checkQueue();
    }
```

```
    SoftReference<VeryBig> s =
      new SoftReference<VeryBig>(new VeryBig("Soft"));
    WeakReference<VeryBig> w =
      new WeakReference<VeryBig>(new VeryBig("Weak"));
    System.gc();
    LinkedList<PhantomReference<VeryBig>> pa =
      new LinkedList<PhantomReference<VeryBig>>();
    for(int i = 0; i < size; i++) {
      pa.add(new PhantomReference<VeryBig>(
        new VeryBig("Phantom " + i), rq));
      System.out.println("Just created: " + pa.getLast());
      checkQueue();
    }
  }
} /* (Da eseguire per visualizzare l'output) *///:~
```

Eseguendo questo programma, noterete che gli oggetti vengono gestiti dal garbage collector anche se potrete ancora accedervi tramite l'oggetto **Reference**; tenete presente che per ottenere il riferimento reale si utilizza il metodo **get()**. Ricordate che potete redirigere l'output del programma in un file di testo, per consultarlo più comodamente.

Noterete anche che l'oggetto **ReferenceQueue** genera sempre una **Reference** contenente un oggetto **null**. Per utilizzare questa funzionalità potete ereditare da una classe di tipo **Reference** e aggiungere altri metodi alla nuova classe.

## WeakHashMap

La libreria dei contenitori dispone di una **Map** speciale per contenere i riferimenti (chiavi) deboli, le cosiddette *weak key*: la **WeakHashMap**. Questa classe è stata progettata per semplificare la creazione delle *canonicalized mapping*: questo tipo di mappatura vi consente di economizzare memoria, creando soltanto un'istanza di un determinato valore. Quando il programma richiede questo valore, ricerca l'oggetto esistente nella mappatura e lo utilizza, invece di crearne uno da zero. Al momento dell'inizializzazione della mappatura possono essere specificati i valori, ma è più probabile che tali valori siano "costruiti" su richiesta.

Trattandosi di una tecnica per economizzare memoria, è opportuno che la **WeakHashMap** permetta al garbage collector di ripulire in modo automatico le chiavi e i valori. Non occorre eseguire nulla di speciale per le chiavi e i valori

che si vogliono collocare nella **WeakHashMap**: essi vengono inglobati automaticamente in riferimenti **WeakReference** dalla mappa stessa. La condizione che attiva il cleanup è che la chiave non sia più in uso, come mostrato di seguito:

```
//: containers/CanonicalMapping.java
// Dimostrazione della WeakHashMap.
import java.util.*;

class Element {
  private String ident;
  public Element(String id) { ident = id; }
  public String toString() { return ident; }
  public int hashCode() { return ident.hashCode(); }
  public boolean equals(Object r) {
    return r instanceof Element &&
      ident.equals(((Element)r).ident);
  }
  protected void finalize() {
    System.out.println("Finalizing " +
      getClass().getSimpleName() + " " + ident);
  }
}

class Key extends Element {
  public Key(String id) { super(id); }
}

class Value extends Element {
  public Value(String id) { super(id); }
}

public class CanonicalMapping {
  public static void main(String[] args) {
    int size = 1000;
    // Oppure selezionate le dimensioni dalla riga di comando:
    if(args.length > 0)
      size = new Integer(args[0]);
```

```
    Key[] keys = new Key[size];
    WeakHashMap<Key,Value> map =
      new WeakHashMap<Key,Value>();
    for(int i = 0; i < size; i++) {
      Key k = new Key(Integer.toString(i));
      Value v = new Value(Integer.toString(i));
      if(i % 3 == 0)
        keys[i] = k; // Salva come "veri" riferimenti
      map.put(k, v);
    }
    System.gc();
  }
} /* (Da eseguire per visualizzare l'output) *///:~
```

La classe **Key** deve avere i metodi **hashCode()** ed **equals()**, dal momento che viene utilizzata come chiave in una struttura dati "hashed". Il metodo **hashCode()** è stato descritto in precedenza nel capitolo.

Quando eseguirete il programma vedrete che il garbage collector salterà ogni terza chiave (...8, 7, 5, 4, 2, 1), perché un normale riferimento a quella chiave è stato inserito anche nell'array **keys**, i cui elementi non possono essere oggetto di garbage collection.

## Contenitori di Java 1.0 e Java 1.1

Molto codice è stato scritto utilizzando i contenitori di Java 1.0 e Java 1.1, e talvolta persino il nuovo codice viene scritto inglobando queste classi. Così, anche se non dovreste mai utilizzare i vecchi contenitori quando scrivete nuovo codice, è comunque necessario che ne conosciate il funzionamento. Tenete presente che le funzionalità dei vecchi contenitori erano abbastanza limitate, pertanto l'autore ha ritenuto di ridurre l'argomento all'essenziale; tenuto conto che si tratta di oggetti ormai anacronistici, ha cercato anche di astenersi dall'enfatizzare eccessivamente alcune delle decisioni progettuali spaventose che li hanno caratterizzati.

### *Vector ed Enumeration*

L'unica sequenza a espansione automatica disponibile in Java 1.0 e Java 1.1 era la classe **Vector**, che quindi è stata utilizzata diffusamente in passato. I difetti di **Vector** sono numerosi, e non è opportuno segnalarli dettagliatamente

in queste pagine: se siete interessati all'argomento, potete consultare la prima edizione di questo libro, scaricabile gratuitamente dal sito *www.mindview.net*.

In pratica, potete considerare **Vector** come un **ArrayList** con nomi di metodo lunghi e poco pratici. Nella revisione della libreria di contenitori Java, **Vector** è stata adattata in modo da funzionare come una **Collection** e una **List**. Questa scelta si è rivelata poco felice perché potrebbe indurre il programmatore a ritenere che le funzionalità di **Vector** siano migliorate, mentre in realtà la sua esistenza è giustificata unicamente per ragioni di supporto al vecchio codice Java.

Per le versioni Java 1.0 e Java 1.1 è stato deciso di inventare un nuovo nome per l'iteratore, rinominandolo come *enumeration* (enumerazione), invece di continuare a utilizzare il termine, per molti già familiare, di "iteratore". L'interfaccia **Enumeration** è più ridotta rispetto a quella di **Iterator**, possiede solo due metodi e usa nomi di metodo più lunghi: il metodo **boolean hasMoreElements()** restituisce **true** se l'enumerazione contiene ulteriori elementi, e **Objectnext-Element()** restituisce l'elemento successivo dell'enumerazione corrente (se presente); se non sono disponibili ulteriori elementi, viene sollevata un'eccezione.

**Enumeration** è soltanto un'interfaccia, non un'implementazione vera e propria, riscontrabile ancora oggi in alcune nuove librerie. Anche se nel vostro codice, quando possibile, dovreste utilizzare un **Iterator**, dovete essere preparati ad affrontare eventuali librerie che vogliano passarvi un'**Enumeration**.

Potete generare un'**Enumeration** per ogni **Collection** ricorrendo al metodo **Collections.enumeration()**, come mostra l'esempio seguente:

```
//: containers/Enumerations.java
// Vector ed Enumeration di Java 1.0 e Java 1.1.
import java.util.*;
import net.mindview.util.*;

public class Enumerations {
  public static void main(String[] args) {
    Vector<String> v =
      new Vector<String>(Countries.names(10));
    Enumeration<String> e = v.elements();
    while(e.hasMoreElements())
      System.out.print(e.nextElement() + ", ");
    // Produce una Enumeration da una Collection:
    e = Collections.enumeration(new ArrayList<String>());
  }
```

```
} /* Output:
ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO, BURUNDI,
CAMEROON, CAPE VERDE, CENTRAL AFRICAN REPUBLIC, CHAD,
*///:~
```

In questo esempio, per creare un'**Enumeration** viene chiamato il metodo **elements()**; l'**Enumeration** può quindi essere utilizzata per eseguire una iterazione in avanti.

Nell'ultima riga di codice viene creato un **ArrayList** e utilizzato il metodo **enumeration()** per adattare un'**Enumeration** dall'**Iterator ArrayList**. Come vedete, quindi, se avete codice vecchio che necessita di interagire con **Enumeration** potete anche servirvi dei moderni contenitori.

## *Hashtable*

Nel confronto prestazionale analizzato in precedenza avete visto che la classe **Hashtable** è molto simile ad **HashMap**, persino nei nomi di metodo. Pertanto, non c'è alcun motivo per continuare a usare **Hashtable** nel nuovo codice.

## *Stack*

Il concetto di stack è stato introdotto in precedenza a proposito della **LinkedList**. Ciò che è strano nella classe **Stack** di Java 1.0 e Java 1.1 è il fatto che, invece di utilizzare un **Vector** tramite composizione, **Stack** *eredita* da **Vector**. **Stack** ha dunque tutte le caratteristiche e i comportamenti di un oggetto **Vector**, oltre ad alcune funzionalità aggiuntive tipiche di **Stack**. È difficile capire se all'epoca i progettisti di Sun ritenessero veramente che questo fosse il modo adatto di progettare questa classe, o se si sia trattato soltanto di un'ingenuità di progettazione; rimane il fatto che la classe **Stack** non è stata rivista con attenzione prima di essere posta in distribuzione, cosicché questo componente è ancora in circolazione anche se nessuno lo utilizza più.

Di seguito, è presentata una semplice dimostrazione delle caratteristiche di **Stack** che memorizza ogni rappresentazione di **String** contenuta in una **enum**. Nel codice è anche illustrato come utilizzare altrettanto facilmente come stack una **LinkedList**, oppure la classe **Stack** creata nel Volume 1, Capitolo 11:

```
//: containers/Stacks.java
// Dimostrazione della classe Stack.
import java.util.*;
import static net.mindview.util.Print.*;
```

```
enum Month { JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,
  JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER }

public class Stacks {
  public static void main(String[] args) {
    Stack<String> stack = new Stack<String>();
    for(Month m : Month.values())
      stack.push(m.toString());
    print("stack = " + stack);
    // Come trattare uno stack come Vector:
    stack.addElement("Ultima riga");
    print("element 5 = " + stack.elementAt(5));
    print("popping elements:");
    while(!stack.empty())
      printnb(stack.pop() + " ");

    // Uso di una LinkedList come Stack:
    LinkedList<String> lstack = new LinkedList<String>();
    for(Month m : Month.values())
      lstack.addFirst(m.toString());
    print("lstack = " + lstack);
    while(!lstack.isEmpty())
      printnb(lstack.removeFirst() + " ");

    // Uso della classe Stack creata nel capitolo 11:
    net.mindview.util.Stack<String> stack2 =
      new net.mindview.util.Stack<String>();
    for(Month m : Month.values())
      stack2.push(m.toString());
    print("stack2 = " + stack2);
    while(!stack2.empty())
      printnb(stack2.pop() + " ");

  }
/* Output:
stack = [JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY,
```

```
AUGUST, SEPTEMBER, OCTOBER, NOVEMBER]
element 5 = JUNE
popping elements:
The last line NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE
MAY APRIL MARCH FEBRUARY JANUARY lstack = [NOVEMBER,
OCTOBER, SEPTEMBER, AUGUST, JULY, JUNE, MAY, APRIL, MARCH,
FEBRUARY, JANUARY]
NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE MAY APRIL MARCH
FEBRUARY JANUARY stack2 = [NOVEMBER, OCTOBER, SEPTEMBER,
AUGUST, JULY, JUNE, MAY, APRIL, MARCH, FEBRUARY, JANUARY]
NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE MAY APRIL MARCH
FEBRUARY JANUARY
*///:~
```

Una rappresentazione di **String** viene creata tramite le costanti **enum Month** e inserita nello **Stack** con il metodo **push()**; in seguito, viene prelevata dalla parte superiore dello stack con il metodo **pop()**. Per maggiore sicurezza, le operazioni di **Vector** vengono anche eseguite sull'oggetto **Stack**: questo è possibile perché, in virtù dell'ereditarietà, uno **Stack** *è* un **Vector**; di conseguenza, tutte le operazioni che possono essere eseguite su **Vector** possono esserlo anche su **Stack**, per esempio il metodo **elementAt()**.

Come accennato in precedenza, se volete implementare la funzionalità di uno stack dovrete utilizzare una **LinkedList**, o la classe **net.mindview.util.Stack** creata a partire dalla classe **LinkedList**.

## *BitSet*

Un **BitSet** viene utilizzato per memorizzare in modo efficiente una grande quantità di dati di tipo binario. È efficiente soltanto dal punto di vista della dimensione: dal punto di vista delle prestazioni, in realtà, **BitSet** si dimostra un po' più lento di un array nativo.

Inoltre, la dimensione minima di un **BitSet** è quella di un **long**: 64 bit. Questo implica che se state memorizzando dati di dimensione inferiore, per esempio valori a 8 bit, troverete dispendioso l'utilizzo di un **BitSet**; se la dimensione è un problema, per contenere i dati è preferibile creare una classe personalizzata, o magari un array. Tenete presente queste due opzioni soltanto se dovete creare *molti* oggetti che contengono elenchi di informazioni di dimensioni ridotte (on/off, 0/1 ecc.) e soltanto dopo avere eseguito i test opportuni e considerato altri parametri. Se decidete di utilizzare queste opzioni solo

per problemi di dimensioni, vi ritroverete alle prese con difficoltà maggiori e sprecherete tempo inutilmente.

Un normale contenitore si espande automaticamente quando vengono aggiunti altri elementi; **BitSet** fa altrettanto. L'esempio seguente mostra il funzionamento della classe **BitSet**:

```
//: containers/Bits.java
// Dimostrazione della classe BitSet.
import java.util.*;
import static net.mindview.util.Print.*;

public class Bits {
  public static void printBitSet(BitSet b) {
    print("bits: " + b);
    StringBuilder bbits = new StringBuilder();
    for(int j = 0; j < b.size() ; j++)
      bbits.append(b.get(j) ? "1" : "0");
    print("bit pattern: " + bbits);
  }
  public static void main(String[] args) {
    Random rand = new Random(47);
    // Prende il valore LSB (least significant bit),
    // cioe' il bit meno significativo di nextInt():
    byte bt = (byte)rand.nextInt();
    BitSet bb = new BitSet();
    for(int i = 7; i >= 0; i--)
      if(((1 << i) &  bt) != 0)
        bb.set(i);
      else
        bb.clear(i);
    print("byte value: " + bt);
    printBitSet(bb);

    short st = (short)rand.nextInt();
    BitSet bs = new BitSet();
    for(int i = 15; i >= 0; i--)
      if(((1 << i) &  st) != 0)
```

```
        bs.set(i);
      else
        bs.clear(i);
    print("short value: " + st);
    printBitSet(bs);

    int it = rand.nextInt();
    BitSet bi = new BitSet();
    for(int i = 31; i >= 0; i--)
      if(((1 << i) &  it) != 0)
        bi.set(i);
      else
        bi.clear(i);
    print("int value: " + it);
    printBitSet(bi);

    // Test di bitset >= 64 bits:
    BitSet b127 = new BitSet();
    b127.set(127);
    print("set bit 127: " + b127);
    BitSet b255 = new BitSet(65);
    b255.set(255);
    print("set bit 255: " + b255);
    BitSet b1023 = new BitSet(512);
    b1023.set(1023);
    b1023.set(1024);
    print("set bit 1023: " + b1023);
  }
} /* Output:
byte value: -107
bits: {0, 2, 4, 7}
bit pattern:
10101001000000000000000000000000000000000000000000000000000
00000
short value: 1302
bits: {1, 2, 4, 8, 10}
```

```
bit pattern:
01101000101000000000000000000000000000000000000000000000000
00000
int value: -2014573909
bits: {0, 1, 3, 5, 7, 9, 11, 18, 19, 21, 22, 23, 24, 25,
26, 31}
bit pattern:
11010101010100000011011111100001000000000000000000000000000
00000
set bit 127: {127}
set bit 255: {255}
set bit 1023: {1023, 1024}
*///:~
```

Nel codice viene utilizzato il generatore di numeri casuali per creare un valore **byte**, uno **short** e un **int**, ciascuno dei quali viene poi convertito in rappresentazione binaria all'interno di un **BitSet**. Questa operazione funziona benissimo perché un **BitSet** è di 64 bit, pertanto nessuno dei suddetti valori ne incrementa la dimensione. Sono poi creati alcuni **BitSet** di dimensioni superiori: come potete notare, il **BitSet** è espandibile secondo le esigenze.

Un **EnumSet** (di cui si parlerà nel Capitolo 7) rappresenta solitamente una scelta migliore rispetto a un **BitSet** quando avete un insieme fisso di flag cui potete assegnare un nome, in quanto **EnumSet** consente di gestire nomi anziché rappresentazioni binarie, riducendo così le possibilità di errori. **EnumSet** impedisce anche l'aggiunta involontaria di nuove posizioni di flag, che potrebbe provocare alcuni bug di difficile individuazione. Gli unici motivi per cui dovreste utilizzare **BitSet** in luogo di **EnumSet** sono la mancata conoscenza del numero di flag necessari in fase di esecuzione, l'impossibilità di assegnare nomi ai flag, oppure la necessità di usufruire di funzionalità specifiche di **BitSet**: in ogni caso, prima di operare una scelta, consultate la documentazione JDK per **BitSet** e **EnumSet**.

## Riepilogo

La libreria di contenitori è indiscutibilmente la più importante di ogni linguaggio orientato agli oggetti; la maggior parte dei programmatori, infatti, si serve dei contenitori più di qualsiasi altro componente di libreria. Alcuni linguaggi, per esempio Python, arrivano a includere i contenitori fondamentali (quali liste, mappe e set) direttamente come parte del linguaggio.

Come viene spiegato nel Volume 1, Capitolo 11, i contenitori consentono di utilizzare facilmente alcune funzionalità molto interessanti. Tuttavia, per usufruirne nel migliore dei modi, a un certo punto sarete costretti a studiarli a fondo: in particolare, dovete approfondire l'argomento "hash" per scrivere metodi **hashCode()** personalizzati; dovrete anche conoscere le varie implementazioni dei contenitori, per scegliere quello che meglio si adatta alle vostre necessità. In questo capitolo questi concetti sono stati analizzati, unitamente a una serie di dettagli aggiuntivi che vi saranno preziosi nell'utilizzo della libreria di contenitori. A questo punto dovreste essere sufficientemente preparati per impiegare i contenitori Java nella vostra attività quotidiana di programmatori.

La progettazione di una libreria è sempre un compito complesso, e questo è ovviamente vero anche per le librerie di contenitori. In C++, le classi contenitore sono alla base di molte classi eterogenee: per quanto riguarda la disponibilità una situazione migliore di quella precedente l'avvento delle classi contenitore di C++, che era nulla. Tuttavia questo concetto di miglioramento non è stato "tradotto" altrettanto bene in Java. All'altro estremo, si possono trovare librerie di contenitori composte di una sola classe, "contenitore", che agisce sia come sequenza lineare sia come array associativo. La libreria di contenitori Java si pone in perfetto equilibrio tra le funzionalità complete che vi attendete da una libreria di contenitori "matura" e la facilità di apprendimento e di utilizzo che le classi contenitore di C++ e altre librerie analoghe non consentono di ottenere. Il risultato può presentare singolarità che molti potrebbero giudicare "strane". A differenza di alcune decisioni che caratterizzano le prime librerie di Java, queste peculiarità non sono casuali ma frutto di decisioni accuratamente ponderate, basate su compromessi tra funzionalità e complessità.

*La soluzione degli esercizi è disponibile nel documento The Thinking in Java Annotated Solution Guide, in vendita all'indirizzo www.mindview.net.*

# Capitolo 6

# Input/Output

La creazione di un buon sistema di input/output (I/O) è uno dei compiti più difficili per il progettista di un linguaggio, come dimostrano i numerosi approcci possibili.

Apparentemente la difficoltà consiste nel tenere conto di tutte le possibilità, perché non soltanto occorre considerare le diverse origini e destinazioni di I/O con cui comunicare (file, console, connessioni di rete ecc.), ma è anche necessario saper dialogare in un'ampia varietà di modi (sequenziale, casuale, bufferizzato, binario, carattere, per righe, per parole ecc.).

I progettisti delle librerie Java hanno affrontato questo problema creando diverse classi. In effetti, sebbene il numero di classi destinate al sistema di input/output di Java sia tale da intimidire il programmatore, ironicamente il sistema di I/O è stato concepito per impedire la proliferazione delle classi. Occorre anche tenere in considerazione le importanti modifiche introdotte nella libreria I/O dalle versioni successive a Java 1.0, quando alla libreria "**byte** oriented" originale sono state aggiunte librerie di I/O "**char** oriented" e dotate di supporto Unicode.

Le classi **nio** sono state integrate nell'intento di migliorare le prestazioni e le funzionalità: l'acronimo **nio** indica "new I/O", un nome introdotto nel JDK 1.4 e ormai in utilizzo da così tanti anni da essere diventato, in qualche modo, "vec-

chio". È essenziale che comprendiate il funzionamento di un certo numero di classi prima di poter utilizzare correttamente i meccanismi di I/O di Java. È anche importante che comprendiate l'evoluzione della libreria di I/O, anche se la vostra prima reazione potrebbe essere: "Non mi interessa la sua evoluzione, voglio soltanto sapere come si utilizza!". Il problema è che senza le necessarie informazioni storiche rischiereste di trovarvi rapidamente confusi in merito ad alcune classi, trovando difficile capire quando servirsene e quando evitarle.

Questo capitolo vi fornirà un'introduzione alle varie classi di I/O della libreria standard di Java e vi mostrerà come utilizzarle al meglio.

## Classe File

Prima di affrontare le classi che effettivamente leggono e scrivono i flussi di dati, è opportuno esaminare una classe di utilità che vi sarà di aiuto nella manipolazione di file e directory.

Il nome della classe **File** è ingannevole, perché all'apparenza sembra far riferimento a un file anche se non è così: in effetti, "FilePath" sarebbe stato un nome più adatto. **File** può rappresentare il *nome* di un file particolare oppure i *nomi* di un insieme di file presenti in una directory. Quando rappresenta un insieme di file, questa classe vi consente di richiedere tale insieme utilizzando il metodo **list()**, che restituisce un array di **String**. In questo caso è più corretto restituire un array invece di una classe contenitore (più flessibile), perché in un array il numero di elementi è fisso; se desiderate ottenere l'elenco dei file presenti in un'altra directory, non dovete fare altro che creare un nuovo oggetto **File**. In questo paragrafo viene mostrato un esempio di impiego di questa classe e dell'interfaccia **FilenameFilter** a essa associata.

### *Ottenere l'elenco di directory*

Supponete di volere visualizzare il contenuto di una directory. A questo scopo potete utilizzare l'oggetto **File** in due modi diversi, innanzitutto chiamando il metodo **list()** senza argomenti per ottenere il contenuto completo dell'oggetto **File**. Se invece desiderate un elenco limitato, per esempio dei soli file con estensione **.java**, dovrete ricorrere a un "filtro di directory" (*directory filter*), vale a dire a una classe che consente la selezione degli oggetti **File** da visualizzare.

Osservate l'esempio seguente, in cui il risultato è stato disposto in ordine alfabetico adottando il metodo **java.util.Arrays.sort()** e il **Comparator String. CASE_INSENSITIVE_ORDER**:

```
//: io/DirList.java
// Visualizza il contenuto di una directory utilizzando
// le espressioni regolari.
// {Args: "D.*\.java"}
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList {
  public static void main(String[] args) {
    File path = new File(".");
    String[] list;
    if(args.length == 0)
      list = path.list();
    else
      list = path.list(new DirFilter(args[0]));
    Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
    for(String dirItem : list)
      System.out.println(dirItem);
  }
}

class DirFilter implements FilenameFilter {
  private Pattern pattern;
  public DirFilter(String regex) {
    pattern = Pattern.compile(regex);
  }
  public boolean accept(File dir, String name) {
    return pattern.matcher(name).matches();
  }
} /* Output:
DirectoryDemo.java
DirList.java
```

```
DirList2.java
DirList3.java
*///:~
```

La classe **DirFilter** implementa l'interfaccia **FilenameFilter**. Notate la semplicità di questa interfaccia:

```
public interface FilenameFilter {
  boolean accept(File dir, String name);
}
```

L'unica ragione che giustifichi l'esistenza di **DirFilter** è fornire il metodo **accept()** al metodo **list()**, in modo che **list()** possa "richiamare" (*call back*) **accept()** per determinare i nomi di directory da includere nella lista: questa è la ragione per cui tale struttura viene spesso chiamata *callback*. In particolare questo è un esempio del design pattern *Strategy*, dove **list()** implementa le funzionalità di base e il programmatore fornisce la "strategia" sotto forma di un **FilenameFilter**, per completare l'algoritmo necessario al funzionamento di **list()**. Dal momento che **list()** accetta come argomento un oggetto **FilenameFilter**, potete passare un oggetto di qualunque classe che implementi **FilenameFilter** per determinare, anche in fase di esecuzione, il comportamento del metodo **list()**. Lo scopo di Strategy è garantire la massima flessibilità nel comportamento del codice.

Il metodo **accept()** deve accettare un oggetto **File** che rappresenta la directory in cui si trova un particolare file, e una **String** che contiene il nome di questo file. Ricordate che il metodo **list()** sta chiamando **accept()** per ciascuno dei nomi di file presenti nell'oggetto directory, per verificare quali possono essere inclusi; il risultato **boolean** restituito da **accept()** indica se un file deve essere incluso nell'elenco. Il metodo **accept()** si serve di un oggetto **matcher** per verificare se l'espressione regolare **regex** corrisponde al nome del file. Servendosi di **accept()**, il metodo **list()** restituisce un array.

### Classi interne anonime

Questo esempio è ideale per riscrivere la funzionalità servendovi di una classe interna, descritta nel Volume 1, Capitolo 10. Per una prima versione, create un metodo **filter()** che restituisce un riferimento a **FilenameFilter**:

```
//: io/DirList2.java
// Utilizzo delle classi interne anonime.
// {Args: "D.*\.java"}
```

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

public class DirList2 {
  public static FilenameFilter filter(final String regex) {
    // Creazione di una classe interna anonima:
    return new FilenameFilter() {
      private Pattern pattern = Pattern.compile(regex);
      public boolean accept(File dir, String name) {
        return pattern.matcher(name).matches();
      }
    }; // Fine della classe interna anonima
  }
  public static void main(String[] args) {
    File path = new File(".");
    String[] list;
    if(args.length == 0)
      list = path.list();
    else
      list = path.list(filter(args[0]));
    Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
    for(String dirItem : list)
      System.out.println(dirItem);
  }
} /* Output:
DirectoryDemo.java
DirList.java
DirList2.java
DirList3.java
*///:~
```

Notate che l'argomento di **filter()** deve essere **final**: questa condizione è necessaria per utilizzare la classe interna al di fuori del proprio ambito.

Questo progetto rappresenta un miglioramento poiché la classe **FilenameFilter** ora è strettamente connessa a **DirList2**; tuttavia, questo approccio vi consente di effettuare un passo ulteriore e definire la classe interna anoni-

ma come argomento di **list()**, nel qual caso il codice risulterà ancora più compatto:

```
//: io/DirList3.java
// Come costruire la classe interna anonima "sul posto".
// {Args: "D.*\.java"}
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList3 {
  public static void main(final String[] args) {
    File path = new File(".");
    String[] list;
    if(args.length == 0)
      list = path.list();
    else
      list = path.list(new FilenameFilter() {
        private Pattern pattern = Pattern.compile(args[0]);
        public boolean accept(File dir, String name) {
          return pattern.matcher(name).matches();
        }
      });
    Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
    for(String dirItem : list)
      System.out.println(dirItem);
  }
} /* Output:
DirectoryDemo.java
DirList.java
DirList2.java
DirList3.java
*///:~
```

Ora l'argomento di **main()** è di tipo **final**, dal momento che la classe interna anonima utilizza direttamente **args[0]**.

Questo dimostra che le classi interne anonime consentono la creazione di classi "usa e getta" specifiche per risolvere determinati problemi. Un vantaggio di questo approccio risiede nella possibilità di mantenere il codice associato a un problema specifico in un punto particolare. D'altra parte tale codice non è sempre di facile lettura, quindi dovreste utilizzare questa tecnica con attenzione.

**Esercizio 1** (3) Modificate **DirList.java** o una delle sue varianti in modo che **FilenameFilter** apra e legga ogni file, utilizzando la classe di utilità **net.mindview.util.TextFile** e accettando il file soltanto se questo contiene uno degli argomenti finali digitati sulla riga di comando.

**Esercizio 2** (2) Create una classe chiamata **SortedDirList** con un costruttore che accetta un oggetto **File** e crea un elenco ordinato dei file che sono contenuti in questo oggetto. Aggiungete alla classe due metodi **list()** sovraccarichi: il primo restituirà l'intera lista, mentre il secondo produrrà un sottoinsieme della lista che soddisfa un'espressione regolare fornita come argomento.

**Esercizio 3** (3) Modificate **DirList.java** o una delle sue varianti in modo che calcoli la somma delle dimensioni dei file selezionati.

## *Utility per directory*

Un'attività comune in programmazione è l'esecuzione di operazioni su gruppi di file, sia nella directory locale sia attraversando l'intera alberatura di directory: a questo scopo, vi sarà utile uno strumento che generi questi gruppi di file automaticamente.

La classe di utilità seguente crea un array di oggetti **File** presenti nella directory locale utilizzando il metodo **local()**, oppure una **List<File>** dell'intera alberatura di directory a partire dalla directory specificata, servendosi del metodo **walk()**: tenete presente che gli oggetti **File** sono più utili dei nomi di file, perché contengono una quantità maggiore di informazioni. Le directory vengono scelte in base a un'espressione regolare, che dovrete specificare:

```
//: net/mindview/util/Directory.java
// Genera una sequenza di oggetti File che corrispondono
// a un'espressione regolare, in una directory locale o
// attraversando un'intera alberatura
package net.mindview.util;
import java.util.regex.*;
import java.io.*;
```

```
import java.util.*;

public final class Directory {
  public static File[]
  local(File dir, final String regex) {
    return dir.listFiles(new FilenameFilter() {
      private Pattern pattern = Pattern.compile(regex);
      public boolean accept(File dir, String name) {
        return pattern.matcher(
          new File(name).getName()).matches();
      }
    });
  }
  public static File[]
  local(String path, final String regex) { // Sovraccarico
    return local(new File(path), regex);
  }
  // Un gruppo two-tuple che restituisce una coppia
  // di oggetti:
  public static class TreeInfo implements Iterable<File> {
    public List<File> files = new ArrayList<File>();
    public List<File> dirs = new ArrayList<File>();
    // L'elemento iterabile predefinito è l'elenco di file:
    public Iterator<File> iterator() {
      return files.iterator();
    }
    void addAll(TreeInfo other) {
      files.addAll(other.files);
      dirs.addAll(other.dirs);
    }
    public String toString() {
      return "dirs: " + PPrint.pformat(dirs) +
        "\n\nfiles: " + PPrint.pformat(files);
    }
  }
  public static TreeInfo
  walk(String start, String regex) { // Inizio ripetizione
```

```
    return recurseDirs(new File(start), regex);
  }
  public static TreeInfo
  walk(File start, String regex) { // Sovraccarico
    return recurseDirs(start, regex);
  }
  public static TreeInfo walk(File start) { // Tutto
    return recurseDirs(start, ".*");
  }
  public static TreeInfo walk(String start) {
    return recurseDirs(new File(start), ".*");
  }
  static TreeInfo recurseDirs(File startDir, String regex){
    TreeInfo result = new TreeInfo();
    for(File item : startDir.listFiles()) {
      if(item.isDirectory()) {
        result.dirs.add(item);
        result.addAll(recurseDirs(item, regex));
      } else // File normale
        if(item.getName().matches(regex))
          result.files.add(item);
    }
    return result;
  }
  // Un semplice test di convalida:
  public static void main(String[] args) {
    if(args.length == 0)
      System.out.println(walk("."));
    else
      for(String arg : args)
       System.out.println(walk(arg));
  }
} ///:~
```

Il metodo **local()** si serve di una variante di **File.list()**, chiamata **listFiles()**, che genera un array di **File**; come vedete, utilizza anche **FilenameFilter**. Se vi

occorre una **List** anziché un array, potrete convertire il risultato servendovi di **Arrays.asList()**.

Il metodo **walk()** converte il nome della directory di inizio in un oggetto **File** e chiama **recurseDirs()**, che esegue un'analisi ricorsiva della directory e raccoglie informazioni a ogni ciclo. Per distinguere i normali file dalle directory il valore restituito è una "tupla" di oggetti, una **List** di normali file e una contenente le directory. In questo caso specifico i campi sono stati resi volutamente **public**, perché l'unico scopo di **TreeInfo** è raccogliere gli oggetti. In realtà, se dovete soltanto restituire una **List**, certamente non la renderete **private**. Notate che **TreeInfo** implementa **Iterable<File>** per recuperare i file, cosicché si ha un'"iterazione predefinita" sull'elenco di file, mentre per specificare le directory si utilizza "**.dirs**".

Il metodo **TreeInfo.toString()** si serve di una classe di "formattazione elegante" per rendere più agevole la lettura dell'output: i metodi predefiniti **toString()** per i contenitori, infatti, visualizzano tutti gli elementi su una sola riga. Per collezioni di grandi dimensioni una simile tecnica di visualizzazione può risultare scomoda da leggere, quindi è opportuno ricorrere a una formattazione alternativa. Ecco uno strumento che aggiunge interruzioni di riga e applica un rientro a ogni elemento:

```
//: net/mindview/util/PPrint.java
// Formattatore per collection
package net.mindview.util;
import java.util.*;

public class PPrint {
  public static String pformat(Collection<?> c) {
    if(c.size() == 0) return "[]";
    StringBuilder result = new StringBuilder("[");
    for(Object elem : c) {
      if(c.size() != 1)
        result.append("\n  ");
      result.append(elem);
    }
    if(c.size() != 1)
      result.append("\n");
    result.append("]");
    return result.toString();
```

```
  }
  public static void pprint(Collection<?> c) {
    System.out.println(pformat(c));
  }
  public static void pprint(Object[] c) {
    System.out.println(pformat(Arrays.asList(c)));
  }
} ///:~
```

Il metodo **pformat()** restituisce una **String** formattata da una **Collection** e il metodo **pprint()** utilizza la stringa ottenuta da **pformat()** per svolgere il proprio lavoro. Come vedete, i casi speciali di "nessun elemento" e di "elemento singolo" vengono gestiti diversamente. Esiste anche una versione di **pprint()** per gli array.

La classe di utilità **Directory** si trova nel package **net.mindview.util**, ed è quindi facilmente disponibile. Ecco come potete servirvene:

```
//: io/DirectoryDemo.java
// Esempio di utilizzo della classe Directory.
import java.io.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class DirectoryDemo {
  public static void main(String[] args) {
    // Tutte le directory:
    PPrint.pprint(Directory.walk(".").dirs);
    // Tutti i file che iniziano con 'T':
    for(File file : Directory.local(".", "T.*"))
      print(file);
    print("----------------------");
    // Tutti i file Java che iniziano con 'T':
    for(File file : Directory.walk(".", "T.*\\.java"))
      print(file);
    print("======================");
    // file Class contenenti "Z" o "z":
    for(File file : Directory.walk(".",".*[Zz].*\\.class"))
```

```
      print(file);
  }
} /* Output: (Esempio)
[.\xfiles]
.\TestEOF.class
.\TestEOF.java
.\TransferTo.class
.\TransferTo.java
----------------------
.\TestEOF.java
.\TransferTo.java
.\xfiles\ThawAlien.java
======================
.\FreezeAlien.class
.\GZIPcompress.class
.\ZipCompress.class
*///:~
```

Per comprendere i secondi argomenti passati ai metodi **local()** e **walk()** potreste avere la necessità di ripassare le vostre conoscenze sulle espressioni regolari, trattate nel Volume 1, Capitolo 12.

È possibile compiere un ulteriore passo, creando uno strumento che percorra le alberature di directory *ed* elabori i file che vi sono contenuti secondo le indicazioni dell'oggetto **Strategy**; questo è un altro esempio del design pattern *Strategy*:

```
//: net/mindview/util/ProcessFiles.java
package net.mindview.util;
import java.io.*;

public class ProcessFiles {
  public interface Strategy {
    void process(File file);
  }
  private Strategy strategy;
  private String ext;
  public ProcessFiles(Strategy strategy, String ext) {
```

```
    this.strategy = strategy;
    this.ext = ext;
  }
  public void start(String[] args) {
    try {
      if(args.length == 0)
        processDirectoryTree(new File("."));
      else
        for(String arg : args) {
          File fileArg = new File(arg);
          if(fileArg.isDirectory())
            processDirectoryTree(fileArg);
          else {
            // Permette all'utente di togliere l'estensione:
            if(!arg.endsWith("." + ext))
              arg += "." + ext;
            strategy.process(
              new File(arg).getCanonicalFile());
          }
        }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
  public void
  processDirectoryTree(File root) throws IOException {
    for(File file : Directory.walk(
        root.getAbsolutePath(), ".*\\." + ext))
      strategy.process(file.getCanonicalFile());
  }
  // Dimostra come utilizzare questa classe:
  public static void main(String[] args) {
    new ProcessFiles(new ProcessFiles.Strategy() {
      public void process(File file) {
        System.out.println(file);
      }
```

```
    }, "java").start(args);
  }
} /* (Da eseguire per visualizzare l'output) *///:~
```

L'interfaccia **Strategy** è nidificata all'interno di **ProcessFiles**: per implementarla dovete specificare **implement ProcessFiles.Strategy**, che fornirà maggiore contesto al lettore del vostro codice. **ProcessFiles** esegue il lavoro di ricerca dei file caratterizzati da una particolare estensione, rappresentata dall'argomento **ext** del costruttore; quando trova un file corrispondente si limita a passarlo all'oggetto **Strategy**, anch'esso passato come argomento al costruttore.

In assenza di argomenti, **ProcessFiles** presupporrà che vogliate percorrere tutte le directory che fanno capo a quella corrente. Potete anche specificare un determinato file (con o senza estensione) che verrà eventualmente aggiunto, oppure una o più directory.

In **main()** potete vedere un semplice esempio di utilizzo di questo strumento, che visualizza i nomi di tutti i file sorgente Java in base ai parametri forniti dalla riga di comando.

> **Esercizio 4** (2) Utilizzate il metodo **Directory.walk()** per sommare le dimensioni di tutti i file in un'alberatura di directory; i nomi dei file dovranno corrispondere a un'espressione regolare.
>
> **Esercizio 5** (1) Modificate **ProcessFiles.java** in modo che ricerchi i file in base a un'espressione regolare, anziché utilizzare un'estensione fissa.

## *Verificare e creare directory*

La classe **File** non si limita alla rappresentazione di file o directory. Potete anche utilizzare un oggetto **File** per creare una nuova directory o un intero percorso se questo non esiste, nonché per visualizzare le caratteristiche dei file (dimensione, data di ultima modifica, lettura/scrittura), verificare se un oggetto **File** rappresenta un file o una directory e cancellare un file. L'esempio seguente mostra come servirsi di alcuni dei metodi disponibili nella classe **File**; per l'elenco completo dei metodi disponibili potete consultare la documentazione JDK, all'indirizzo *http://java.sun.com*.

```
//: io/MakeDirectories.java
// Dimostra l'utilizzo della classe File per creare directory
// e manipolare file.
// {Args: MakeDirectoriesTest}
```

```
import java.io.*;

public class MakeDirectories {
  private static void usage() {
    System.err.println(
      "Usage: MakeDirectories path1 ...\n" +
      "Creates each path\n" +
      "Usage: MakeDirectories -d path1 ...\n" +
      "Delete each path\n" +
      "Usage: MakeDirectories -r path1 path2\n" +
      "Rename from path1 to path2");
    System.exit(1);
  }
  private static void fileData(File f) {
    System.out.println(
      "Absolute path: " + f.getAbsolutePath() +
      "\n canRead: " + f.canRead() +
      "\n canWrite: " + f.canWrite() +
      "\n getName: " + f.getName() +
      "\n getParent: " + f.getParent() +
      "\n getPath: " + f.getPath() +
      "\n length: " + f.length() +
      "\n lastModified: " + f.lastModified());
    if(f.isFile())
      System.out.println("It's a file");
    else if(f.isDirectory())
      System.out.println("It's a directory");
  }
  public static void main(String[] args) {
    if(args.length < 1) usage();
    if(args[0].equals("-r")) {
      if(args.length != 3) usage();
      File
        old = new File(args[1]),
        rname = new File(args[2]);
      old.renameTo(rname);
```

```
      fileData(old);
      fileData(rname);
      return; // Exit main
    }
    int count = 0;
    boolean del = false;
    if(args[0].equals("-d")) {
      count++;
      del = true;
    }
    count--;
    while(++count < args.length) {
      File f = new File(args[count]);
      if(f.exists()) {
        System.out.println(f + " exists");
        if(del) {
          System.out.println("deleting..." + f);
          f.delete();
        }
      }
      else { // Non esiste
        if(!del) {
          f.mkdirs();
          System.out.println("created " + f);
        }
      }
      fileData(f);
    }
  }
} /* Output: (80% match)
Created MakeDirectoriesTest
Absolute path: /Users/marcomac/Progetti/Thinking in Java/IO/
build/classes/MakeDirectoriesTest
 canRead: true
 canWrite: true
 getName: MakeDirectoriesTest
 getParent: null
```

```
  getPath: MakeDirectoriesTest
  length: 0
  lastModified: 1101690308831
It's a directory
*///:~
```

In **fileData()** potete osservare i vari metodi di ricerca dei file utilizzati per visualizzare informazioni relative a un file o a un percorso di directory.

Il primo metodo utilizzato in **main()** è **renameTo()**, che vi consente di rinominare (o spostare) un file in un percorso completamente diverso, specificato come argomento; anche questo nuovo percorso è un oggetto **File**. Questo metodo funziona anche con nomi di directory di qualsiasi lunghezza.

Eseguendo alcuni esperimenti con il programma precedente, noterete che è possibile creare un percorso di directory di qualsiasi livello di complessità, perché il metodo **mkdirs()** eseguirà tutto il lavoro automaticamente.

> **Esercizio 6** (5) Utilizzate la classe **ProcessFiles** per trovare tutti i file sorgente Java che sono stati modificati dopo una certa data all'interno di una particolare sottodirectory.

## Input e output

Le librerie di I/O dei linguaggi di programmazione ricorrono spesso al concetto di *flusso* (*stream*), che rappresenta qualsiasi origine o destinazione dei dati sotto forma di oggetti in grado di produrre o ricevere dati. Il flusso nasconde i dettagli funzionali interni della periferica di I/O reale.

Osservando la gerarchia di classi nella documentazione JDK, noterete che le classi della libreria Java per l'I/O sono suddivise nelle categorie di input e output. Grazie all'ereditarietà, tutti gli oggetti che derivano dalle classi **InputStream** o da **Reader** possiedono metodi di base chiamati **read()** per la lettura di un singolo **byte** o di un array di **byte**. Analogamente, tutte le classi che derivano da **OutputStream** o **Writer** hanno metodi di base **write()** per scrivere un singolo **byte** o un array di **byte**. Di norma, tuttavia, non vi servirete di questi metodi, che vengono utilizzati quasi esclusivamente da altre classi dotate a loro volta di interfacce più pratiche. Pertanto, raramente creerete il vostro oggetto di flusso servendovi di una singola classe, ma di preferenza stratificherete più oggetti per fornire alla vostra classe le funzionalità opportune; questa strategia di progettazione è conforme al design pattern *Decorator*, come vedrete nel prosieguo del capitolo. Il fatto che occorra creare più oggetti per generare un singolo flusso è la ragione principale per cui la libreria di I/O di Java appare confusa.

È utile classificare le classi in base alle funzionalità. In Java 1.0, i progettisti delle librerie avevano deciso che tutte le classi dedicate all'input dovessero ereditare da **InputStream** e quelle che gestiscono l'output da **OutputStream**.

Come d'abitudine in questo manuale, le varie classi saranno adeguatamente descritte: è tuttavia necessario che facciate riferimento alla documentazione JDK per approfondirne i dettagli, in particolare l'elenco completo dei metodi di ogni classe.

## *Tipi di InputStream*

Lo scopo della classe **InputStream** è rappresentare le classi che producono l'input da fonti differenti, vale a dire:

1. Un array di byte.
2. Un oggetto **String**.
3. Un file.
4. Una "pipe" (pipa), che funziona come un tubo fisico: gli oggetti entrano a un'estremità ed escono dall'altra.
5. Una sequenza di altri flussi, da raccogliere in un flusso singolo.
6. Altre fonti, quali una connessione Internet. Questo argomento è trattato in *Thinking in Enterprise Java*, disponibile su *www.mindview.net*.

Ogni fonte è associata a una sottoclasse di **InputStream**. Inoltre, la classe **FilterInputStream** è anch'essa di tipo **InputStream** e rappresenta la classe di base per le classi "decorator", utili per collegare attributi o interfacce ai flussi di input. Questo argomento sarà trattato in seguito.

**Tabella I/O-1. Tipi di InputStream**

| *Classe* | *Funzione* | *Argomenti del costruttore e loro utilizzo* |
|---|---|---|
| **ByteArray-InputStream** | Consente l'utilizzo di un buffer in memoria come **InputStream**. | Il buffer da cui estrarre i byte. |
| | | **Come origine dei dati**: da collegare a un oggetto **FilterInputStream** per fornire un'interfaccia utile. |

**Tabella I/O-1. Tipi di InputStream**

| *Classe* | *Funzione* | *Argomenti del costruttore e loro utilizzo* |
|---|---|---|
| **StringBuffer-InputStream** | Converte una **String** in **InputStream**. | Una **String**. L'implementazione sottostante utilizza effettivamente uno **StringBuffer**. |
| | | **Come origine dei dati**: da collegare a un oggetto **FilterInputStream** per fornire un'interfaccia utile. |
| **FileInputStream** | Per leggere informazioni da un file. | Una **String** che rappresenta il nome di un file, o un oggetto **File** o **FileDescriptor**. |
| | | **Come origine dei dati**: da collegare a un oggetto **FilterInputStream** per fornire un'interfaccia utile. |
| **PipedInputStream** | Genera i dati che devono essere scritti sulla **PipedOutputStream** associata. Implementa il concetto di "pipe". | **PipedOutputStream**. |
| | | **Come origine dei dati nel multithreading**: da collegare a un oggetto **FilterInputStream** per fornire un'interfaccia utile. |
| **Sequence-InputStream** | Converte due o più oggetti **InputStream** in un solo **InputStream**. | Due oggetti **InputStream** o un'**Enumeration** per un contenitore di oggetti **InputStream**. |
| | | **Come origine dei dati**: da collegare a un oggetto **FilterInputStream** per fornire un'interfaccia utile. |
| **FilterInputStream** | È una classe astratta che rappresenta un'interfaccia per le classi "decorator", e fornisce utili funzionalità alle altre classi **InputStream**. Si veda la Tabella I/O-3. | Si veda la Tabella I/O-3. |
| | | Si veda la Tabella I/O-3. |

## *Tipi di OutputStream*

Questa categoria include le classi che stabiliscono dove verrà scritto il vostro output: un array di byte, un file o una "pipe"; non una **String**, però, benché possiate crearne una dall'array di byte.

Inoltre, **FilterOutputStream** fornisce una classe di base per le classi "decorator", utili per associare attributi o interfacce ai flussi di output. Questo argomento sarà trattato in seguito.

**Tabella I/O-2. Tipi di OutputStream**

| ***Classe*** | ***Funzione*** | ***Argomenti del costruttore e loro utilizzo*** |
|---|---|---|
| **ByteArray-OutputStream** | Crea un buffer in memoria; tutti i dati trasmessi al flusso vengono inseriti in questo buffer. | Dimensione iniziale facoltativa del buffer. |
| | | **Per indicare la destinazione dei dati**: da collegare a un oggetto **FilterOutputStream** per fornire un'interfaccia utile. |
| **FileOutputStream** | Per la trasmissione di informazioni a un file. | Una **String** che rappresenta il nome di un file, o un oggetto **File** o **FileDescriptor**. |
| | | **Per indicare la destinazione dei dati**: da collegare a un oggetto **FilterOutputStream** per fornire un'interfaccia utile. |
| **Piped-OutputStream** | Tutte le informazioni inviate a questo oggetto vengono automaticamente trasformate in input per il **PipedInputStream** associato. Implementa il concetto di "pipe". | **PipedInputStream**. |
| | | **Per indicare la destinazione dei dati nel multithreading**: da collegare a un oggetto **FilterOutputStream** per fornire un'interfaccia utile. |
| **Filter-OutputStream** | È una classe astratta che rappresenta un'interfaccia per le classi "decorator" e fornisce utili funzionalità alle altre classi **OutputStream**. Si veda la Tabella I/O-4. | Si veda la Tabella I/O-4. |
| | | Si veda la Tabella I/O-4. |

## Aggiunta di attributi e interfacce utili

Il design pattern Decorator è stato introdotto nel Capitolo 3, nel paragrafo "Utilizzo del modello Decorator". La libreria di I/O di Java richiede molte combinazioni di caratteristiche diverse, e per questo motivo è giustificato il ricorso al modello Decorator.[1]

Il motivo che legittima l'esistenza delle classi "filtro" nella libreria di I/O di Java è la classe astratta "filter", che è quella di base di tutti i decorator. Un "decoratore" deve avere la stessa interfaccia dell'oggetto che "decora" ma può anche estendere l'interfaccia, come avviene in numerose classi "filtro".

Tuttavia, il pattern Decorator presenta un inconveniente. I decoratori vi offrono una maggiore flessibilità in fase di scrittura dei programmi poiché vi consentono di mescolare a vostro piacimento gli attributi, ma rendono anche complesso il codice. Il motivo per cui la libreria di I/O di Java è poco pratica da utilizzare è che richiede la creazione di numerose classi (il tipo I/O principale, più tutti i decoratori) per ottenere il singolo oggetto di I/O che desiderate. Le classi che forniscono l'interfaccia di "decorazione" per gestire un **InputStream** o un **OutputStream** sono **FilterInputStream** e **FilterOutputStream**, nomi decisamente poco intuitivi. Le classi **FilterInputStream** e **FilterOutputStream** sono derivate dalle classi di base della libreria I/O, **InputStream** e **OutputStream**; questo è un requisito essenziale del decoratore, al fine di fornire un'interfaccia comune a tutti gli oggetti da decorare.

### *Leggere da un InputStream con FilterInputStream*

Le classi **FilterInputStream** eseguono due operazioni molto diverse. **DataInputStream** vi consente di leggere i vari tipi di dati primitivi e gli oggetti **String**: tutti i metodi iniziano con "read", come **readByte()**, **readFloat()** ecc. La classe **DataInputStream** e la classe **DataOutputStream** associata vi consentono di spostare i dati primitivi da una posizione a un'altra tramite un flusso; queste "posizioni" sono determinate dalle classi elencate nella Tabella I/O-1.

Le rimanenti classi **FilterInputStream** modificano il comportamento interno di un **InputStream**: se il flusso è bufferizzato oppure no, se tiene traccia delle righe lette (per visualizzare il numero di riga o impostarlo) e se si può "respingere" (*push back*) un determinato carattere. Le ultime due classi ricordano da vicino i supporti per la costruzione di un compilatore: è possibile

1. Non è chiaro se questa sia stata una decisione progettuale valida, soprattutto tenendo conto della semplicità delle librerie di I/O di altri linguaggi: in ogni caso, questa è la motivazione che giustifica tale scelta.

che siano state aggiunte per supportare l'esperimento di "creazione di un compilatore Java in Java", e probabilmente non le utilizzerete nella vostra attività di programmazione quotidiana.

A prescindere dalla periferica di I/O cui vi state collegando, dovrete quasi sempre bufferizzare il vostro input, quindi sarebbe stato più sensato integrare nella libreria di I/O il caso speciale o una semplice chiamata di metodo per l'input non bufferizzato, invece di quello per l'input bufferizzato.

| **Tabella I/O-3. Tipi di FilterInputStream** | | |
|---|---|---|
| ***Classe*** | ***Funzione*** | ***Argomenti del costruttore e loro utilizzo*** |
| **Data-InputStream** | Utilizzata in abbinamento a **DataOutputStream**, consente la lettura dei primitivi (**int**, **char**, **long**, ecc.) da un flusso, con una modalità portabile. | **InputStream**.<br>Contiene un'interfaccia completa per leggere i tipi primitivi. |
| **Buffered-InputStream** | Utilizzate questa classe per evitare la lettura fisica ogni volta che vi occorrono altri dati; equivale all'utilizzo di un buffer. | **InputStream**, con dimensione del buffer facoltativa.<br>Non possiede un'interfaccia autonoma, ma si limita ad aggiungere il buffering al processo. Da collegare a un oggetto che fornisca un'interfaccia. |
| **LineNumber-InputStream** | Tiene traccia dei numeri di riga nel flusso; consente di chiamare i metodi **getLineNumber()** e **setLineNumber(int)**. | **InputStream**.<br>Si limita ad aggiungere i numeri di riga, quindi dovrebbe essere collegata a un oggetto che fornisca un'interfaccia. |
| **Pushback-InputStream** | Possiede un buffer di un byte che "respinge" (*push back*) l'ultimo carattere letto. | **InputStream**.<br>Utilizzata di norma nella scansione eseguita da un compilatore. Il suo utilizzo da parte del programmatore è alquanto improbabile. |

## Scrivere su un OutputStream con FilterOutputStream

Il naturale complemento di **DataInputStream** è la classe **DataOutputStream**, la quale formatta ogni tipo primitivo e gli oggetti **String** in un flusso che potrà essere letto da qualsiasi **DataInputStream**, su qualunque computer. Tutti i nomi di metodo iniziano con "write": **writeByte()**, **writeFloat()** ecc.

L'intento originale di **PrintStream** era quello di produrre tutti i tipi di dati primitivi e gli oggetti **String** in un formato visualizzabile. Questo è un approccio diverso da quello di **DataOutputStream**, il cui l'obiettivo è inserire gli elementi dei dati in un flusso, in modo che **DataInputStream** possa ricostruirli con una modalità portabile.

I due metodi importanti nella classe **PrintStream** sono **print()** e **println()**, sovraccaricati per visualizzare tutti i vari tipi: la differenza tra **print()** e **println()** è che quest'ultimo metodo aggiunge automaticamente un simbolo di nuova riga o "a capo".

La classe **PrintStream** può essere problematica da gestire, poiché intercetta tutte le **IOException**: richiede di testare esplicitamente la condizione d'errore con **checkError()**, che restituisce **true** qualora si verifichi un errore. Inoltre, **PrintStream** non è internazionalizzabile correttamente e non gestisce la fine della riga in modo indipendente dalla piattaforma. Questi problemi vengono risolti da **PrintWriter**, descritta in seguito.

La classe **BufferedOutputStream** è un modificatore il quale indica al flusso di utilizzare il buffer, in modo da non dare luogo a una scrittura fisica ogni volta che viene scritto il flusso. Probabilmente utilizzerete sempre questa tecnica per produrre i vostri output.

**Tabella I/O-4. Tipi di FilterOutputStream**

| ***Classe*** | ***Funzione*** | ***Argomenti del costruttore e loro utilizzo*** |
|---|---|---|
| **Data-OutputStream** | Utilizzata in abbinamento a **DataInputStream**, consente la scrittura dei primitivi (**int**, **char**, **long** ecc.) in un flusso, con una modalità portabile. | **OutputStream**.<br>Contiene un'interfaccia completa per scrivere i tipi primitivi. |

| Tabella I/O-4. Tipi di FilterOutputStream | | |
|---|---|---|
| *Classe* | *Funzione* | *Argomenti del costruttore e loro utilizzo* |
| **PrintStream** | Per produrre output formattato. Mentre **DataOutputStream** gestisce la memorizzazione dei dati, **PrintStream** si occupa della visualizzazione. | **OutputStream**, con valore **boolean** facoltativo per indicare se il buffer deve essere svuotato dopo ogni nuova riga. |
| | | Dovrebbe essere il wrapping "finale" dell'oggetto **OutputStream**. Il suo utilizzo da parte del programmatore è alquanto frequente. |
| **Buffered-OutputStream** | Utilizzate questa classe per evitare la scrittura fisica ogni volta che volete produrre altri dati; equivale all'utilizzo di un buffer. Per svuotare il buffer, potete chiamare il metodo **flush()**. | **OutputStream**, con dimensione del buffer facoltativa. |
| | | Non possiede un'interfaccia autonoma, ma si limita ad aggiungere al processo la messa in buffer. Da collegare a un oggetto che fornisca un'interfaccia. |

## Classi Reader e Writer

Java 1.1 ha apportato modifiche significative alla libreria di base dei flussi di I/O. Osservando le classi **Writer** e **Reader** il vostro primo pensiero (come quello dell'autore) potrebbe essere che siano state ideate per sostituire le classi **OutputStream** e **InputStream**, ma non è così. Sebbene alcuni aspetti della libreria di flussi originale siano disapprovati e provochino un avvertimento da parte del compilatore, le classi **InputStream** e **OutputStream** offrono importanti funzionalità sotto forma di I/O orientato ai byte (*byte oriented*); le classi **Writer** e **Reader** forniscono invece I/O di tipo carattere (*character based*), compatibile con lo standard Unicode. Inoltre:

1. Java 1.1 ha aggiunto nuove classi nella gerarchia di **OutputStream** e **InputStream**, quindi è evidente che non è prevista la sostituzione di queste classi.
2. Vi sono situazioni in cui è necessario utilizzare le classi della gerarchia "byte" *in abbinamento* a quelle della gerarchia "carattere". Per rendere possibile questa combinazione sono disponibili le cosiddette classi *adapter*, che

fungono da adattatori: **InputStreamReader** converte un **InputStream** in un **Reader** e **OutputStreamWriter** converte un **OutputStream** in un **Writer**.

L'internazionalizzazione è il motivo principale che giustifica l'esistenza delle gerarchie **Writer** e **Reader**. La vecchia gerarchia di flussi I/O supporta soltanto flussi byte di 8 bit e non gestisce adeguatamente i caratteri a 16 bit del formato Unicode. Poiché l'internazionalizzazione si basa su questo formato e il tipo **char** nativo di Java è di tipo Unicode a 16 bit, in Java 1.1 sono state aggiunte le gerarchie **Reader** e **Writer** che supportano il formato Unicode in tutte le operazioni di I/O. Inoltre, le nuove librerie sono state progettate per eseguire tali operazioni più velocemente.

## *Origini e destinazioni dei dati*

Quasi tutte le classi originali dei flussi I/O di Java possiedono corrispondenti classi **Reader** e **Writer** che garantiscono la gestione nativa del formato Unicode. Tuttavia, in alcune situazioni le classi byte-oriented **InputStream** e **OutputStream** sono la soluzione corretta; in particolare, le librerie **java.util.zip** sono di tipo byte-oriented anziché char-oriented. Di conseguenza, l'approccio più ragionevole è *provare* a utilizzare le classi **Writer** e **Reader** ogni volta che sia possibile. Scoprirete da soli quali situazioni richiedono le librerie byte-oriented, poiché il vostro codice non compilerà. La tabella seguente mostra la corrispondenza tra le origini e le destinazioni fisiche dei dati per le due gerarchie.

| ***Origini e destinazioni: classi Java 1.0*** | ***Classi Java 1.1 corrispondenti*** |
|---|---|
| **InputStream** | **Reader**<br>adattatore:<br>**InputStreamReader** |
| **OutputStream** | **Writer**<br>adattatore:<br>**OutputStreamWriter** |
| **FileInputStream** | **FileReader** |
| **FileOutputStream** | **FileWriter** |
| **StringBufferInputStream** (deprecata) | **StringReader** |
| (nessuna classe corrispondente) | **StringWriter** |
| **ByteArrayInputStream** | **CharArrayReader** |
| **ByteArrayOutputStream** | **CharArrayWriter** |
| **PipedInputStream** | **PipedReader** |
| **PipedOutputStream** | **PipedWriter** |

In generale, scoprirete che le interfacce delle due gerarchie sono simili, anche se non identiche.

## *Modificare il comportamento dei flussi*

Per **InputStream** e **OutputStream**, i flussi sono stati adattati alle necessità particolari utilizzando le sottoclassi "decorator" **FilterInputStream** e **FilterOutputStream**. Le gerarchie di classi **Writer** e **Reader** seguono un concetto analogo, ma non esattamente lo stesso.

Nella tabella seguente, la corrispondenza è una grezza approssimazione di quella precedente; la differenza è dovuta all'organizzazione delle classi: sebbene **BufferedOutputStream** sia una sottoclasse di **FilterOutputStream**, **BufferedWriter** *non è* una sottoclasse di **FilterWriter**: pur essendo di tipo **abstract** non ha sottoclassi, quindi sembra essere stata inserita come segnaposto o semplicemente per fare in modo che non vi chiediate dove sia. Tuttavia, tra le interfacce e le classi esiste una notevole corrispondenza.

| ***Filtri: classi Java 1.0*** | ***Classi Java 1.1 corrispondenti*** |
|---|---|
| **FilterInputStream** | **FilterReader** |
| **FilterOutputStream** | **FilterWriter** (classe astratta senza sottoclassi) |
| **FilterInputStream** | **FilterReader** |
| **FilterOutputStream** | **FilterWriter** (classe astratta senza sottoclassi) |
| **BufferedInputStream** | **BufferedReader** (dispone anche di **readLine()**) |
| **BufferedOutputStream** | **BufferedWriter** |
| **DataInputStream** | Utilizzare **DataInputStream** (tranne se è necessario utilizzare **readLine()**, nel qual caso vi servirete della classe **BufferedReader**) |
| **PrintStream** | **PrintWriter** |
| **LineNumberInputStream** (deprecata) | **LineNumberReader** |
| **StreamTokenizer** | **StreamTokenizer** (Utilizzare il costruttore che accetta un oggetto **Reader**) |
| **PushbackInputStream** | **PushbackReader** |

Le direttive sono abbastanza chiare: ogni volta che volete ricorrere al metodo **readLine()** non dovreste utilizzare **DataInputStream**, con il quale otterreste

un messaggio "deprecation warning" in fase di compilazione, ma preferibilmente un **BufferedReader**. A parte questo, **DataInputStream** è pur sempre un membro "privilegiato" della libreria di I/O.

Per semplificare la transizione all'utilizzo di **PrintWriter** vengono forniti alcuni costruttori che, oltre agli oggetti **Writer**, accettano qualsiasi oggetto **OutputStream**. L'interfaccia di formattazione di **PrintWriter** è praticamente la stessa di **PrintStream**.

Java SE5 ha dotato **PrintWriter** di costruttori per facilitare la creazione di file al momento di eseguire l'output, che vedrete tra breve.

Un costruttore di **PrintWriter** ha anche un'opzione per eseguire automaticamente il flush, che si attiva dopo ogni **println()** se l'apposito flag del costruttore è stato impostato.

### *Classi non modificate*

Alcune classi, elencate nella tabella seguente, non hanno subito modifiche nella transizione da Java 1.0 a Java 1.1.

| *Classi Java 1.0 senza corrispondenze con classi Java 1.1* |
|---|
| **DataOutputStream** |
| **File** |
| **RandomAccessFile** |
| **SequenceInputStream** |

La classe **DataOutputStream**, in particolare, essendo utilizzata senza modifiche, richiede l'adozione delle gerarchie di **OutputStream** e **InputStream** per registrare e leggere dati in modo trasportabile.

## Classe RandomAccessFile

**RandomAccessFile** è utilizzato per i file contenenti record di dimensione conosciuta, in modo che possiate spostarvi da un record a un altro servendovi del il metodo **seek()**, per leggerli o modificarli. I record non devono necessariamente avere le stesse dimensioni: dovete soltanto determinare la loro dimensione e posizione nel file.

A una prima occhiata, è difficile credere che **RandomAccessFile** non appartenga alle gerarchie **OutputStream** o **InputStream**, in ogni caso non ha al-

tre analogie con queste gerarchie tranne l'implementazione delle interfacce **DataOutput** e **DataInput**, peraltro implementate anche da **DataInputStream** e **DataOutputStream**. **RandomAccessFile** inoltre non utilizza nessuna delle funzionalità delle classi **OutputStream** o **InputStream**: è una classe totalmente separata, scritta da zero con metodi propri, la maggior parte dei quali nativi. Il motivo può risiedere nel fatto che **RandomAccessFile** ha un comportamento fondamentalmente diverso rispetto agli altri tipi di I/O, dal momento che vi permette di spostarvi avanti e indietro in un file. In ogni caso è una classe "solitaria", diretta discendente di **Object**.

In un certo senso, **RandomAccessFile** funziona come un flusso **DataInputStream** abbinato a un **DataOutputStream**, con i metodi **getFilePointer()** per individuare la posizione all'interno del file, **seek()** per spostarsi in un punto specifico del file e **lenght()** per determinare la dimensione massima del file. Inoltre i costruttori richiedono un secondo argomento, come **fopen()** in C, che indica se state eseguendo una semplice lettura casuale ("**r**") oppure operazioni di lettura e scrittura ("**rw**"). Non è tuttavia disponibile il supporto per i file in sola lettura.

I metodi di ricerca sono disponibili unicamente in **RandomAccessFile**, che funziona soltanto con i file. La classe **BufferedInputStream** vi consente di contrassegnare con **mark()** una particolare posizione, il cui valore viene mantenuto in una variabile interna. È anche possibile ritornare a questa posizione con **reset()**, ma questa funzionalità è limitata e non particolarmente utile.

A partire dal JDK 1.4 la maggior parte, se non tutte, le funzionalità di **RandomAccessFile** sono state sostituite dai file *memory mapped* **nio**, che saranno trattati nel prosieguo del capitolo.

## Utilizzi tipici dei flussi di I/O

Anche se è possibile combinare i flussi I/O in modi diversi, probabilmente vi servirete soltanto di un numero limitato di combinazioni; potete utilizzare i programmi seguenti come esempi tipici delle librerie di I/O.

In questi listati la gestione delle eccezioni è stata semplificata inviando le eccezioni a console, una tecnica adatta unicamente nei piccoli programmi d'esempio e di utilità. Nel vostro codice dovreste utilizzare tecniche di gestione degli errori più elaborate.

### *Bufferizzazione dei file di input*

Per aprire un file per l'input di caratteri, utilizzate un **FileInputReader** con un oggetto **String** oppure un **File** come nome di file. Al fine di ottimizzare la

velocità è opportuno che il file sia bufferizzato, passando il riferimento risultante al costruttore di **BufferedReader**. Poiché **BufferedReader** dispone anche del metodo **readLine()**, questo sarà il vostro oggetto finale e l'interfaccia da cui eseguire la lettura. Quando **readLine()** restituisce il valore **null**, significa che vi trovate al termine del file.

```
//: io/BufferedInputFile.java
import java.io.*;

public class BufferedInputFile {
  // Solleva le eccezioni alla console:
  public static String
  read(String filename) throws IOException {
    // Lettura dell'input riga per riga:
    BufferedReader in = new BufferedReader(
      new FileReader(filename));
    String s;
    StringBuilder sb = new StringBuilder();
    while((s = in.readLine())!= null)
      sb.append(s + "\n");
    in.close();
    return sb.toString();
  }
  public static void main(String[] args)
  throws IOException {
    System.out.print(read("BufferedInputFile.java"));
  }
} /* (Da eseguire per visualizzare l'output) *///:~
```

L'oggetto **StringBuilder sb** viene utilizzato per accumulare l'intero contenuto del file, inclusi i caratteri di nuova riga (*newline*), che tuttavia devono essere aggiunti esplicitamente perché vengono eliminati dal metodo **readLine()**. Infine, viene chiamato il metodo **close()** per chiudere il file.[2]

2. Nel progetto originale **close()** avrebbe dovuto intervenire all'esecuzione del metodo **finalize()**, ed è appunto questo il modo in cui **finalize()** è definito per le classi di I/O. Tuttavia, come si è visto in precedenza nel manuale, **finalize()** non funziona come i progettisti di Java avevano previsto: per dirla in termini più chiari, non funziona affatto. Pertanto, l'unico approccio sicuro consiste nel chiamare esplicitamente **close()** per chiudere i file.

**Esercizio 7** (2) Aprite un file di testo in modo da leggerlo una riga alla volta come **String**, e inserite questo oggetto **String** in una **LinkedList**. Visualizzate poi in ordine inverso tutte le righe contenute nella **LinkedList**.

**Esercizio 8** (1) Modificate l'Esercizio 7 in modo che il nome del file sia fornito come argomento da riga di comando.

**Esercizio 9** (1) Modificate l'Esercizio 8 per trasformare in maiuscolo tutte le righe contenute nella **LinkedList**, e inviate i risultati a **System.out**.

**Esercizio 10** (2) Modificate l'Esercizio 8 affinché accetti parole da ricercare nel file come argomenti supplementari da riga di comando. Visualizzate tutte le righe in cui sono presenti le parole trovate.

**Esercizio 11** (2) Nell'esempio **innerclasses/GreenhouseController.java**, nel codice della classe **GreenhouseController** è registrato un insieme di eventi. Modificate il programma in modo che tali eventi e la loro durata vengano letti da un file di testo. Livello di difficoltà 8: utilizzate un design pattern *Factory Method* per costruire gli eventi. Per maggiori dettagli, consultate *Thinking in Patterns (with Java)* all'indirizzo *www.mindview.net.*

## *Input dalla memoria*

Questo esempio si serve della **String** ottenuta da **BufferedInputFile.read()** per creare uno **StringReader**; quindi il metodo **read()** legge tutti i caratteri, uno per volta, e li visualizza a console:

```
//: io/MemoryInput.java
import java.io.*;

public class MemoryInput {
  public static void main(String[] args)
  throws IOException {
    StringReader in = new StringReader(
      BufferedInputFile.read("MemoryInput.java"));
    int c;
    while((c = in.read()) != -1)
      System.out.print((char)c);
  }
} /* (Da eseguire per visualizzare l'output) *///:~
```

Notate che **read()** restituisce il carattere successivo come **int**; questo deve poi venire convertito in **char** per essere visualizzato correttamente.

## Formattazione dell'input dalla memoria

Per leggere dati "formattati" si utilizza **DataInputStream**, che è una classe di I/O di tipo byte-oriented; di conseguenza, occorre utilizzare le classi **InputStream** in luogo delle classi **Reader**. Naturalmente, con le classi **InputStream** potete leggere qualsiasi cosa sotto forma di byte (per esempio, un file), ma in questo caso specifico viene utilizzata una **String**:

```
//: io/FormattedMemoryInput.java
import java.io.*;

public class FormattedMemoryInput {
  public static void main(String[] args)
  throws IOException {
    try {
      DataInputStream in = new DataInputStream(
        new ByteArrayInputStream(
         BufferedInputFile.read(
          "FormattedMemoryInput.java").getBytes()));
      while(true)
        System.out.print((char)in.readByte());
    } catch(EOFException e) {
      System.err.println("End of stream");
    }
  }
} /* (Da eseguire per visualizzare l'output) *///:~
```

Un flusso **ByteArrayInputStream** richiede che gli sia passato un array di byte; per creare tale array, servitevi del metodo **getBytes()** della classe **String**. L'oggetto **ByteArrayInputStream** risultante è un **InputStream** che può essere passato a **DataInputStream**.

Se leggete i caratteri da un **DataInputStream** un byte per volta utilizzando il metodo **readByte()**, ogni valore **byte** sarà un risultato legittimo, pertanto il valore restituito non potrà servire per rilevare la fine dell'input. In alternativa, per sapere quanti altri caratteri sono disponibili potete impiegare il metodo **available()**. L'esempio seguente mostra come leggere un file un byte per volta:

```
//: io/TestEOF.java
// Test di fine file, in lettura "un byte alla volta".
import java.io.*;
```

```
public class TestEOF {
  public static void main(String[] args)
  throws IOException {
    DataInputStream in = new DataInputStream(
      new BufferedInputStream(
        new FileInputStream("TestEOF.java")));
    while(in.available() != 0)
      System.out.print((char)in.readByte());
  }
} /* (Da eseguire per visualizzare l'output) *///:~
```

Notate che **available()** funziona in modo diverso a seconda del tipo di supporto letto; questo metodo restituisce testualmente "il numero di byte che possono essere letti *senza provocare blocchi*". Per un file questo equivale all'intero documento, ma con un altro tipo di flusso il valore restituito potrebbe essere diverso. Usate questo metodo con attenzione.

In casi come questi, potete rilevare la fine dell'input anche intercettando un'eccezione: tenete presente, tuttavia, che l'utilizzo delle eccezioni per controllare qualsiasi flusso operativo è considerato un abuso di questa funzionalità.

## *Nozioni fondamentali sui file di output*

Un oggetto **FileWriter** scrive i dati in un file e, di norma, l'output viene bufferizzato inglobandolo in un **BufferedWriter**. Se provate a rimuovere il buffer per verificare l'effetto sulle prestazioni, vedrete che la bufferizzazione aumenterà in modo considerevole le prestazioni dell'I/O. Questo esempio è stato "decorato" con **PrintWriter** per ottenere la formattazione dell'output. I file creati con questa tecnica possono essere letti come normali file di testo:

```
//: io/BasicFileOutput.java
import java.io.*;

public class BasicFileOutput {
  static String file = "BasicFileOutput.out";
  public static void main(String[] args)
  throws IOException {
    BufferedReader in = new BufferedReader(
      new StringReader(
        BufferedInputFile.read("BasicFileOutput.java")));
```

```
    PrintWriter out = new PrintWriter(
      new BufferedWriter(new FileWriter(file)));
    int lineCount = 1;
    String s;
    while((s = in.readLine()) != null )
      out.println(lineCount++ + ": " + s);
    out.close();
    // Visualizza il file archiviato:
    System.out.println(BufferedInputFile.read(file));
  }
} /* (Da eseguire per visualizzare l'output) *///:~
```

Via via che le righe vengono scritte, nel file è aggiunto anche il numero di riga; notate che *non* viene utilizzata la classe **LineNumberReader**, perché poco opportuna. Come vedete nell'esempio, è molto semplice tenere traccia dei numeri di riga.

Al termine del flusso di input il metodo **readLine()** restituisce **null**; a questo punto, occorre chiamare in modo esplicito **close()** per **out**: ricordate che se non chiamate **close()** su tutti i file di output i buffer potrebbero non essere svuotati, pertanto il file risulterà incompleto.

### Variante abbreviata per l'output su file di testo

Java SE5 ha integrato un costruttore di supporto a **PrintWriter**, in modo che non occorra generare manualmente i decoratori ogni volta che si deve creare un file di testo per scrivervi. Nell'esempio seguente, **BasicFileOutput.java** è stato riscritto per utilizzare questa forma abbreviata:

```
//: io/FileOutputShortcut.java
import java.io.*;

public class FileOutputShortcut {
  static String file = "FileOutputShortcut.out";
  public static void main(String[] args)
  throws IOException {
    BufferedReader in = new BufferedReader(
      new StringReader(
       BufferedInputFile.read("FileOutputShortcut.java")));
```

```
    // Ecco la variante abbreviata:
    PrintWriter out = new PrintWriter(file);
    int lineCount = 1;
    String s;
    while((s = in.readLine()) != null )
      out.println(lineCount++ + ": " + s);
    out.close();
    // Visualizza il file archiviato:
    System.out.println(BufferedInputFile.read(file));
  }
} /* (Da eseguire per visualizzare l'output) *///:~
```

Come potete vedere, la bufferizzazione avviene comunque, pur senza dovere occuparvene direttamente. Purtroppo non sono state previste abbreviazioni per altre comuni operazioni, di conseguenza il tipico codice di I/O implicherà una considerevole ridondanza. In ogni caso la classe di utilità **TextFile**, utilizzata in questo manuale e definita nel prosieguo del capitolo, semplifica queste operazioni comuni.

**Esercizio 12** (3) Modificate l'Esercizio 8 per aprire un file di testo in scrittura. Registrate nel file le righe contenute nella **LinkedList**, comprensive di numeri di riga, e senza ricorrere alle classi "LineNumber".

**Esercizio 13** (3) Modificate **BasicFileOutput.java** in modo che utilizzi **LineNumberReader** per tenere traccia dei numeri di riga. Notate che è molto più facile da gestire rispetto alla gestione dei numeri di riga che si ottiene scrivendo del codice apposito.

**Esercizio 14** (2) Basandovi su **BasicFileOutput.java**, scrivete un programma che confronta le prestazioni di scrittura su un file, utilizzando l'I/O bufferizzato e quello non bufferizzato.

## *Memorizzazione e recupero dei dati*

La classe **PrintWriter** formatta i dati in modo che siano leggibili da una persona. Al fine di produrre dati che possano essere recuperati da un altro flusso, utilizzerete un **DataOutputStream** per scrivere i dati e un **DataInputStream** per recuperarli.

Naturalmente questi flussi possono essere di qualsiasi tipo; l'esempio seguente si serve di un file, bufferizzato in lettura e in scrittura. **DataOutputStream**

e **DataInputStream** sono byte-oriented, quindi richiedono **InputStream** e **OutputStream**:

```
//: io/StoringAndRecoveringData.java
import java.io.*;

public class StoringAndRecoveringData {
  public static void main(String[] args)
  throws IOException {
    DataOutputStream out = new DataOutputStream(
      new BufferedOutputStream(
        new FileOutputStream("Data.txt")));
    out.writeDouble(3.14159);
    out.writeUTF("That was pi");
    out.writeDouble(1.41413);
    out.writeUTF("Square root of 2");
    out.close();
    DataInputStream in = new DataInputStream(
      new BufferedInputStream(
        new FileInputStream("Data.txt")));
    System.out.println(in.readDouble());
    // Solo il metodo readUTF() è in grado di recuperare
    // adeguatamente le stringhe Java-UTF:
    System.out.println(in.readUTF());
    System.out.println(in.readDouble());
    System.out.println(in.readUTF());
  }
} /* Output:
3.14159
That was pi
1.41413
Square root of 2
```

Quando utilizzate un flusso **DataOutputStream** per scrivere i dati, potete recuperarli in modo corretto servendovi di un **DataInputStream**, a prescindere dalle piattaforme adottate per leggere e scrivere i dati. Ciò è molto importante, tenuto conto che ogni programmatore sa quanto tempo richiede la soluzione dei problemi legati alle caratteristiche specifiche delle varie piat-

taforme. Questo problema non esiste se su entrambi i computer è installato Java.[3]

Quando utilizzate un **DataOutputStream**, l'unico modo sicuro per scrivere una **String** che sia recuperabile con un **DataInputStream** risiede nell'utilizzo della codifica UTF-8; nell'esempio che segue, questa operazione è eseguita per mezzo dei metodi **writeUTF()** e **readUTF()**.

UTF-8 è una codifica multibyte la cui lunghezza varia in base al set di caratteri in utilizzo. Se state operando (in tutto o in parte) con caratteri ASCII, che occupano solo sette bit, Unicode costituirà un enorme spreco di spazio e di ampiezza di banda, poiché codifica i caratteri ASCII in un solo byte e quelli non ASCII in due o tre byte. Inoltre, la lunghezza della stringa è memorizzata nei primi due byte della stringa UTF-8. Tuttavia **writeUTF()** e **readUTF()** si servono di una variante speciale della codifica UTF-8, descritta nella documentazione JDK per questi metodi; pertanto, per leggere correttamente una stringa scritta con **writeUTF()** utilizzando un programma non Java, dovrete realizzare codice apposito.

Con **writeUTF()** e **readUTF()** potete mescolare **String** e altri tipi di dati utilizzando un flusso **DataOutputStream**, sapendo che le stringhe verranno memorizzate in formato Unicode e saranno facilmente recuperabili con un **DataInputStream**.

Il metodo **writeDouble**() scrive i numeri **double** nel flusso e il suo metodo complementare **readDouble()** li recupera; esistono metodi simili per leggere e scrivere gli altri tipi di dato. Affinché tali metodi funzionino correttamente dovete conoscere la posizione esatta dell'elemento di dati nel flusso, considerato che un **double** registrato come semplice sequenza di byte potrebbe essere letto come **double**, come **char** ecc. Occorre quindi un formato fisso per i dati da registrare nel file, oppure è necessario memorizzare informazioni aggiuntive nel file che analizzate, mediante le quali determinare la posizione dei dati. Tenete presente che la serializzazione degli oggetti o l'utilizzo del formato XML, entrambi descritti nel prosieguo del capitolo, possono rivelarsi tecniche migliori per registrare e recuperare strutture dati complesse.

**Esercizio 15** (4) Consultate la documentazione JDK per **DataOutputStream** e **DataInputStream** quindi, iniziando da **StoringAndRecoveringData.java**, create un programma che registri e recuperi tutti i tipi di dato possibili mediante le classi **DataOutputStream** e **DataInputStream**. Verificate che i valori siano memorizzati ed estratti in modo corretto.

---

3. XML è un altro modo per risolvere il problema dello spostamento di dati tra piattaforme eterogenee, che non richiede la presenza di Java sui diversi computer. L'utilizzo del linguaggio XML sarà trattato nel prosieguo di questo capitolo.

## Lettura e scrittura di file ad accesso casuale

L'utilizzo della classe **RandomAccessFile** equivale a quello dei flussi **DataInputStream** e **DataOutputStream** abbinati, poiché implementa le stesse interfacce **DataInput** e **DataOutput**; mette inoltre a disposizione il metodo **seek()** per spostarsi nel file e modificare i valori.

Quando utilizzate **RandomAccessFile**, dovete conoscere la struttura del file al fine di manipolarlo correttamente. La classe **RandomAccessFile** possiede metodi specifici per leggere e scrivere i tipi primitivi e le stringhe UTF-8. Ecco un esempio:

```
//: io/UsingRandomAccessFile.java
import java.io.*;

public class UsingRandomAccessFile {
  static String file = "rtest.dat";
  static void display() throws IOException {
    RandomAccessFile rf = new RandomAccessFile(file, "r");
    for(int i = 0; i < 7; i++)
      System.out.println(
        "Value " + i + ": " + rf.readDouble());
    System.out.println(rf.readUTF());
    rf.close();
  }
  public static void main(String[] args)
  throws IOException {
    RandomAccessFile rf = new RandomAccessFile(file, "rw");
    for(int i = 0; i < 7; i++)
      rf.writeDouble(i*1.414);
    rf.writeUTF("The end of the file");
    rf.close();
    display();
    rf = new RandomAccessFile(file, "rw");
    rf.seek(5*8);
    rf.writeDouble(47.0001);
    rf.close();
    display();
  }
```

```
} /* Output:
Value 0: 0.0
Value 1: 1.414
Value 2: 2.828
Value 3: 4.242
Value 4: 5.656
Value 5: 7.069999999999999
Value 6: 8.484
The end of the file
Value 0: 0.0
Value 1: 1.414
Value 2: 2.828
Value 3: 4.242
Value 4: 5.656
Value 5: 47.0001
Value 6: 8.484
The end of the file
*///:~
```

Il metodo **display()** apre un file e visualizza sette elementi come valori **double**. In **main()** il file viene creato, poi aperto e modificato. Poiché un **double** è sempre di otto byte, al fine di recuperare con **seek()** il **double** numero 5, è sufficiente moltiplicare **5*8** per ottenere la posizione da fornire a questo metodo.

Come si è detto, la classe **RandomAccessFile** è considerata separata dal resto della gerarchia di I/O, tranne per il fatto che implementa le interfacce **DataOutput** e **DataInput**. Non supportando i decoratori, non può essere combinata con le funzionalità delle sottoclassi di **InputStream** e **OutputStream**. Dovete anche dare per scontato che un **RandomAccessFile** sia bufferizzato correttamente, poiché non potete integrarvi questa caratteristica.

L'unica opzione di cui disponete è il secondo argomento del costruttore, che vi permette di aprire un **RandomAccessFile** in sola lettura (“**r**”) o in lettura e scrittura (“**rw**”).

In alternativa a **RandomAccessFile**, in effetti, potreste valutare la possibilità di utilizzare i file **nio** di tipo *memory mapped*.

**Esercizio 16** (2) Consultate la documentazione JDK per **RandomAccessFile**. Prendendo come base **UsingRandomAccessFile.java**, create un programma che registri e poi recuperi tutti i tipi di dato possibili

forniti dalla classe **RandomAccessFile**, verificando che i valori siano registrati e caricati in modo accurato.

### *Flussi con pipe*

Fino a questo punto si è soltanto accennato alle classi **PipedInputStream**, **PipedOutputStream**, **PipedReader** e **PipedWriter**. Questo non significa che non siano utili, quanto piuttosto che il loro valore non è avvertibile finché non si inizia a comprendere l'argomento della concorrenza, tenuto conto che i "flussi convogliati" (*piped stream*) sono utilizzati nella comunicazione tra i vari task. Questo argomento sarà trattato con un esempio nel Volume 3, Capitolo 1.

## Utility per la scrittura e la lettura dei file

Un'attività di programmazione molto comune consiste nel caricare un file in memoria, modificarlo e infine salvarlo. Uno dei problemi con la libreria I/O di Java è che l'esecuzione di queste operazioni richiede una quantità elevata di codice: non sono infatti disponibili funzioni di supporto che eseguano queste funzioni automaticamente. Per di più, i decoratori rendono piuttosto difficile ricordare come aprire gli archivi. Pertanto, è opportuno che integriate nella vostra libreria alcune classi di supporto che eseguano in modo più semplice queste operazioni di base. Java SE5 ha aggiunto a **PrintWriter** un pratico costruttore con cui aprire facilmente un file di testo in scrittura. Tuttavia sono molte le operazioni che avrete occasione di eseguire ripetutamente, ed è sempre utile eliminare il codice ridondante connesso a tali mansioni.

Di seguito è mostrata la classe **TextFile** che è stata utilizzata negli esempi precedenti per facilitare la lettura e la scrittura dei file. Questa classe contiene metodi **static** per leggere e scrivere i file di testo come stringa unica, e permette di creare un oggetto **TextFile** che mantiene in un **ArrayList** le righe del file; in questo modo, disponete di tutte le funzionalità di un **ArrayList** pur gestendo in realtà il contenuto di un file:

```
//: net/mindview/util/TextFile.java
// Metodi static per leggere e scrivere file di testo come
// singole stringhe, e per gestire un file come un ArrayList.
package net.mindview.util;
import java.io.*;
import java.util.*;

public class TextFile extends ArrayList<String> {
```

```
// Legge un file come un'unica stringa:
public static String read(String fileName) {
  StringBuilder sb = new StringBuilder();
  try {
    BufferedReader in= new BufferedReader(new FileReader(
      new File(fileName).getAbsoluteFile()));
    try {
      String s;
      while((s = in.readLine()) != null) {
        sb.append(s);
        sb.append("\n");
      }
    } finally {
      in.close();
    }
  } catch(IOException e) {
    throw new RuntimeException(e);
  }
  return sb.toString();
}
// Scrive un singolo file in una chiamata di metodo:
public static void write(String fileName, String text) {
  try {
    PrintWriter out = new PrintWriter(
      new File(fileName).getAbsoluteFile());
    try {
      out.print(text);
    } finally {
      out.close();
    }
  } catch(IOException e) {
    throw new RuntimeException(e);
  }
}
// Legge un file, suddividendolo in base a un'espressione
// regolare qualsiasi:
```

```
public TextFile(String fileName, String splitter) {
  super(Arrays.asList(read(fileName).split(splitter)));
  // L'operazione di split() con espressione regolare spesso
  // lascia una String vuota in prima posizione:
  if(get(0).equals("")) remove(0);
}
// Normalmente legge per riga:
public TextFile(String fileName) {
  this(fileName, "\n");
}
public void write(String fileName) {
  try {
    PrintWriter out = new PrintWriter(
      new File(fileName).getAbsoluteFile());
    try {
      for(String item : this)
        out.println(item);
    } finally {
      out.close();
    }
  } catch(IOException e) {
    throw new RuntimeException(e);
  }
}
// Un semplice test:
public static void main(String[] args) {
  String file = read("TextFile.java");
  write("test.txt", file);
  TextFile text = new TextFile("test.txt");
  text.write("test2.txt");
  // Termina con un elenco ordinato di parole singole:
  TreeSet<String> words = new TreeSet<String>(
    new TextFile("TextFile.java", "\\W+"));
  // Mostra le parole convertite in maiuscolo:
  System.out.println(words.headSet("a"));
}
```

```
} /* Output:
[0, ArrayList, Arrays, Break, BufferedReader, BufferedWriter,
Clean, Display, File, FileReader, FileWriter, Funzioni,
IOException, L, Legge, Mostra, Normally, Normalmente, Output,
PrintWriter, Read, Regular, RuntimeException, Scrive, Simple,
Static, String, StringBuilder, System, TextFile, Tools,
TreeSet, Un, W, Write]
*///:~
```

Il metodo **read()** accoda ogni riga di testo letta a uno **StringBuilder**, aggiungendo un carattere di fine riga che sarà eliminato in fase di lettura, poi restituisce una **String** contenente l'intero file. Il metodo **write()** apre un file e vi scrive il contenuto della **String**.

Notate che tutto il codice che apre un file include la chiamata al metodo **close()** nella clausola **finally**, allo scopo di garantire la corretta chiusura del file.

Il costruttore si serve del metodo **read()** per convertire il file in una **String**, quindi utilizza il metodo **String.split()** per suddividere il risultato in righe, dividendo la **String** in corrispondenza dei caratteri di fine riga: se ritenete di utilizzare questa classe con frequenza, potreste riscrivere il costruttore per migliorarne l'efficienza. Purtroppo Java non prevede un corrispondente metodo di "assemblaggio", pertanto le singole righe devono essere scritte manualmente servendosi del metodo non **static write()**.

Considerato che questa classe è stata progettata per banalizzare il processo di lettura e scrittura dei file, tutte le eccezioni **IOException** sono convertite in **RuntimeException**, evitando così al programmatore client di utilizzare i blocchi **try-catch**. Potreste tuttavia volere creare un'altra versione di questo codice, che passi le **IOException** al metodo chiamante.

All'interno di **main()** viene eseguito un test di base per verificare il funzionamento dei metodi.

Sebbene la creazione di questa classe di utilità non abbia richiesto molto codice, il suo utilizzo vi permetterà di risparmiare una notevole quantità di tempo e semplificherà la vostra attività, come vedrete in altri esempi di questo capitolo.

Un altro modo per risolvere il problema della lettura di file di testo consiste nel ricorrere alla classe **java.util.Scanner**, introdotta da Java SE5; questa classe, tuttavia, è limitata alla lettura dei file, non fa parte di **java.io** ed è progettata soprattutto per la realizzazione di utility di scansione dei linguaggi di programmazione, i cosiddetti "little languages".

**Esercizio 17** (4) Servendovi di **TextFile** e di una **Map<Character, Integer>**, create un programma che conti le occorrenze di tutti i diversi ca-

ratteri presenti in un file: per esempio, se il file contiene 12 occorrenze della lettera "a", il valore **Integer** associato al **Character** che contiene "a" nella **Map** conterrà il valore 12.

**Esercizio 18** (1) Modificate **TextFile.java** in modo che passi le eccezioni **IOException** al chiamante.

## *Lettura di file binari*

Questa classe di utilità è simile a **TextFile.java**, in quanto semplifica la lettura dei file binari:

```
//: net/mindview/util/BinaryFile.java
// Una classe di utilita' per leggere i file in formato
// binario.
package net.mindview.util;
import java.io.*;

public class BinaryFile {
  public static byte[] read(File bFile) throws IOException{
    BufferedInputStream bf = new BufferedInputStream(
      new FileInputStream(bFile));
    try {
      byte[] data = new byte[bf.available()];
      bf.read(data);
      return data;
    } finally {
      bf.close();
    }
  }
  public static byte[]
  read(String bFile) throws IOException {
    return read(new File(bFile).getAbsoluteFile());
  }
} ///:~
```

Un metodo **read()** sovraccarico accetta un argomento **File**, mentre il secondo richiede un argomento **String** che corrisponda al nome del file: entrambi restituiscono un array di **byte**.

Il metodo **available()** è utilizzato per creare l'array della dimensione corretta, che viene poi popolato con il metodo sovraccarico **read()**.

**Esercizio 19** (2) Utilizzando **BinaryFile** e un **Map<Byte, Integer>**, create un programma che conti le occorrenze di tutti i diversi byte in un file.

**Esercizio 20** (4) Servendovi di **Directory.walk()** e **BinaryFile**, verificate che tutti i file **.class** in un'alberatura di directory inizino con i caratteri esadecimali "**CAFEBABE**".

## Standard I/O

Il termine *standard I/O* o input/output standard fa riferimento al concetto UNIX di un unico flusso informativo usato dai programmi, poi adottato in varie forme da Windows e da molti altri sistemi operativi: in accordo a questa concezione, tutto l'input dei programmi proviene da *standard input*, tutto l'output viene inviato a *standard output* e tutti i messaggi di errore a *standard error*. Il valore dello standard I/O è che i programmi possono essere facilmente concatenati, in quanto lo standard output di un programma può diventare lo standard input di un altro: come vedete, si tratta di uno strumento molto potente.

### *Leggere da standard input*

In conformità al modello standard I/O, Java dispone dei flussi **System.in**, **System.out** e **System.err**. In questo manuale avete già visto come scrivere su standard output utilizzando l'oggetto **System.out**, già incluso in ogni oggetto **PrintStream**. Anche **System.err** è analogo a un **PrintStream**, mentre **System.in** è una sorta di **InputStream** grezzo privo della funzionalità di inglobamento (*wrapping*): questo significa che se **System.out** e **System.err** possono essere utilizzati così come sono, **System.in** deve essere incorporato prima di essere utilizzato per la lettura dei dati.

Di norma l'input viene letto una riga per volta con il metodo **readLine()**, e per fare questo occorre inglobare **System.in** in un **BufferedReader**: questo richiede la conversione di **System.in** in **Reader** mediante **InputStreamReader**. Ecco un esempio che visualizza tutto ciò che viene digitato:

```
//: io/Echo.java
// Come leggere da standard input.
// {RunByHand}
import java.io.*;
```

```
public class Echo {
  public static void main(String[] args)
  throws IOException {
    BufferedReader stdin = new BufferedReader(
      new InputStreamReader(System.in));
    String s;
    while((s = stdin.readLine()) != null && s.length()!= 0)
      System.out.println(s);
    // Digitare una riga vuota o Ctrl-Z per terminare
    // il programma
  }
} ///:~
```

Il motivo della presenza della specifica di eccezione è che **readLine()** può sollevare una **IOException**. Tenete presente che di norma **System.in** dovrebbe essere bufferizzato, come avviene per la maggior parte dei flussi.

> **Esercizio 21** (1) Scrivete un programma che legga da standard input e converta in maiuscolo tutti i caratteri, quindi invii i risultati su standard output. Ridirigete il contenuto di un file a questo programma: ricordate che la sintassi per la redirezione varia in funzione del sistema operativo.

## *Convertire System.out in PrintWriter*

**System.out** è un flusso **PrintStream**, che a sua volta è un **OutputStream**. **PrintWriter** possiede un costruttore che accetta come argomento un **OutputStream**; quindi, se lo desiderate, potete convertire **System.out** in un **PrintWriter** ricorrendo a questo costruttore:

```
//: io/ChangeSystemOut.java
// Come convertire System.out in un PrintWriter.
import java.io.*;

public class ChangeSystemOut {
  public static void main(String[] args) {
    PrintWriter out = new PrintWriter(System.out, true);
    out.println("Hello, world!");
  }
```

```
} /* Output:
Hello, world!
*///:~
```

È importante utilizzare la versione del costruttore di **PrintWriter** con due argomenti e impostare il secondo argomento a **true**, per abilitare il *flushing*, vale a dire lo svuotamento automatico del buffer. Se non eseguite questa operazione, potreste non riuscire a visualizzare l'output.

## *Redirezione dello standard I/O*

La classe Java **System** consente di ridirigere i flussi di standard I/O input, output ed error, per mezzo di semplici chiamate a metodi **static**:

**setIn(InputStream)**

**setOut(PrintStream)**

**setErr(PrintStream)**

La redirezione dell'output è particolarmente pratica quando occorra produrre una grande quantità di output a video: in tal caso, infatti, lo scorrimento potrebbe essere troppo rapido per un'agevole lettura delle informazioni.[4]

La redirezione dell'input è utile per i programmi eseguiti da riga di comando, nei quali occorre verificare ripetutamente una particolare sequenza di input fornita dall'utente. Considerate il semplice codice seguente, che mostra come utilizzare questi metodi:

```
//: io/Redirecting.java
// Dimostrazione della ridirezione dello standard I/O.
import java.io.*;

public class Redirecting {
  public static void main(String[] args)
  throws IOException {
    PrintStream console = System.out;
    BufferedInputStream in = new BufferedInputStream(
      new FileInputStream("Redirecting.java"));
    PrintStream out = new PrintStream(
      new BufferedOutputStream(
```

4. Nel Volume 3, Capitolo 2, vedrete una soluzione ancora più pratica: un programma a interfaccia grafica con una casella di testo scorrevole.

```
            new FileOutputStream("test.out")));
        System.setIn(in);
        System.setOut(out);
        System.setErr(out);
        BufferedReader br = new BufferedReader(
          new InputStreamReader(System.in));
        String s;
        while((s = br.readLine()) != null)
          System.out.println(s);
        out.close(); // Ricordatevi di questo!
        System.setOut(console);
    }
} ///:~
```

Questo programma connette lo standard input a un file e ridirige standard output e standard error a un altro file. Notate che all'inizio il programma memorizza il riferimento all'oggetto **System.out** originale, poi ripristinato al termine del programma.

La redirezione dell'I/O manipola i flussi di byte, non quelli di caratteri; pertanto dovete utilizzate gli oggetti **InputStream** e **OutputStream**, invece di **Reader** e **Writer**.

## Controllo dei processi

Spesso dovrete eseguire altri programmi del sistema operativo dall'interno delle vostre applicazioni Java, e controllare l'input e l'output di questi programmi; la libreria di Java mette a disposizione le classi necessarie per eseguire queste operazioni.

Un'operazione comune consiste nell'eseguire un programma e trasmettere l'output risultante al dispositivo di console; in questo paragrafo vedrete una classe di utilità che semplifica questa operazione.

Con questa classe possono verificarsi due tipi di errori: errori normali che generano eccezioni, per i quali verrà sollevato un errore di runtime, ed errori di esecuzione del processo stesso, che devono essere segnalati mediante un'eccezione separata:

```
//: net/mindview/util/OSExecuteException.java
package net.mindview.util;
```

```
public class OSExecuteException extends RuntimeException {
  public OSExecuteException(String why) { super(why); }
} ///:~
```

Per avviare un programma dovete passare al metodo **OSExecute.command()** una stringa **command**, vale a dire le stesse istruzioni che digitereste per eseguire il programma dalla riga di comando. Questo comando viene passato al costruttore di **java.lang.ProcessBuilder**, che lo richiede come sequenza di oggetti **String**; quindi viene avviato l'oggetto risultante **ProcessBuilder**:

```
//: net/mindview/util/OSExecute.java
// Esegue un comando del sistema operativo e invia l'output
// a console.
package net.mindview.util;
import java.io.*;

public class OSExecute {
  public static void command(String command) {
    boolean err = false;
    try {
      Process process =
        new ProcessBuilder(command.split(" ")).start();
      BufferedReader results = new BufferedReader(
        new InputStreamReader(process.getInputStream()));
      String s;
      while((s = results.readLine())!= null)
        System.out.println(s);
      BufferedReader errors = new BufferedReader(
        new InputStreamReader(process.getErrorStream()));
      // Notifica gli errori e restituisce valori diversi
      // da zero per chiamare i processi in caso di problemi:
      while((s = errors.readLine())!= null) {
        System.err.println(s);
        err = true;
      }
    } catch(Exception e) {
```

```
        // Aggiustamento per Windows 2000, che solleva
        // un'eccezione per la riga di comando predefinita:
        if(!command.startsWith("CMD /C"))
          command("CMD /C " + command);
        else
          throw new RuntimeException(e);
      }
      if(err)
        throw new OSExecuteException("Errors executing " +
          command);
    }
} ///:~
```

Per intercettare il flusso di standard output dal programma in esecuzione occorre chiamare il metodo **getInputStream()**, poiché **InputStream** è un oggetto leggibile.

I risultati generati dal programma vengono letti una riga per volta da **readLine()**; a questo punto le righe vengono semplicemente visualizzate, ma potrebbero anche essere intercettate e restituite come risultato della chiamata a **command()**.

Gli errori generati dal programma sono trasmessi al flusso standard error e intercettati chiamando **getErrorStream()**. Gli eventuali errori vengono visualizzati, poi viene sollevata un'eccezione **OSExecuteException** in modo che il programma chiamante possa gestire il problema.

L'esempio seguente mostra come utilizzare **OSExecute**:

```
//: io/OSExecuteDemo.java
// Dimostra la redirezione dello standard I/O.
import net.mindview.util.*;

public class OSExecuteDemo {
  public static void main(String[] args) {
    OSExecute.command("javap OSExecuteDemo");
  }
} /* Output:
Compiled from "OSExecuteDemo.java"
public class OSExecuteDemo extends java.lang.Object{
```

```
    public OSExecuteDemo();
    public static void main(java.lang.String[]);
}
*///:~
```

In questo esempio, per decompilare il programma viene utilizzato il decompilatore **javap** fornito con il JDK.

**Esercizio 22** (5) Modificate **OSExecute.java** in modo che, anziché mostrare il flusso di standard output, restituisca i risultati dell'esecuzione del programma sotto forma di una **List** di **String**. Dimostrate l'impiego di questa nuova versione della classe di utilità.

## Nuova libreria di I/O

La "nuova" libreria di input/output di Java, introdotta con il JDK 1.4 nei pacchetti **java.nio.***, ha un obiettivo primario: la velocità. In effetti, i "vecchi" pacchetti di I/O sono stati implementati *ex novo* servendosi di **nio** per beneficiare di questo incremento di velocità, di conseguenza otterrete comunque alcuni vantaggi anche se non scriverete esplicitamente il codice con **nio**. L'aumento di velocità si avverte sia nell'input/output dei file, analizzato in questo capitolo, sia nell'input/output di rete, che è trattato in *Thinking in Enterprise Java*.

La velocità deriva dall'utilizzo di strutture più prossime alla tecnica adottata dal sistema operativo per gestire l'I/O: i *canali* e i *buffer*. Per analogia, potreste pensare a una miniera di carbone; il canale è la miniera che contiene la vena di carbone (i dati), mentre il buffer è il carrello inviato in miniera, che ritorna pieno di carbone e viene scaricato. In pratica, non interagite direttamente con il canale ma con il buffer, e inviate il buffer nel canale. È il canale che ottiene i dati dal buffer, oppure ve li immette.

L'unico tipo di buffer che comunica direttamente con un canale è il **ByteBuffer**, ovvero un buffer che contiene byte grezzi. Se esaminate la documentazione JDK per **java.nio.ByteBuffer** vedrete che il funzionamento è piuttosto semplice: si crea un buffer specificando la quantità di memoria da allocare e altri metodi si incaricano di inserire o estrarre i dati, nella forma grezza di byte o come tipi di dato primitivi. Tuttavia non vi è modo di inserire o ottenere un oggetto, neppure una **String**. Il meccanismo è di basso livello, per adeguarsi in modo più efficiente alle strutture della maggior parte dei sistemi operativi.

Tre classi nel "vecchio" sistema di I/O sono state modificate al fine di produrre un **FileChannel**: **FileInputStream**, **FileOutputStream** e, sia per lettura sia per scrittura, **RandomAccessFile**. Notate che questi sono flussi di manipolazione dei byte, in armonia con la natura "a basso livello" dell'infrastruttura **nio**. Le classi in modalità carattere **Writer** e **Reader** non producono canali, ma la classe **java.nio.channels.Channels** dispone di metodi pratici per produrre **Reader** e **Writer** dai canali stessi.

Considerate questo semplice esempio che utilizza i tre tipi di flussi per produrre canali soltanto scrivibili, leggibili e scrivibili e soltanto leggibili:

```
//: io/GetChannel.java
// Come ottenere i canali dai flussi
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class GetChannel {
  private static final int BSIZE = 1024;
  public static void main(String[] args) throws Exception {
    // Scrive un file:
    FileChannel fc =
      new FileOutputStream("data.txt").getChannel();
    fc.write(ByteBuffer.wrap("Some text".getBytes()));
    fc.close();
    // Accoda al file:
    fc =
      new RandomAccessFile("data.txt", "rw").getChannel();
    fc.position(fc.size()); // Si sposta al termine del file
    fc.write(ByteBuffer.wrap("; Some more.".getBytes()));
    fc.close();
    // Legge il file:
    fc = new FileInputStream("data.txt").getChannel();
    ByteBuffer buff = ByteBuffer.allocate(BSIZE);
    fc.read(buff);
    buff.flip();
    while(buff.hasRemaining())
      System.out.print((char)buff.get());
```

```
    }
  } /* Output:
  Some text; Some more.
  *///:~
```

Per tutte le classi di flusso illustrate in questo codice, **getChannel()** produrrà un **FileChannel**. Un canale è uno strumento elementare, che accetta un **ByteBuffer** in lettura o in scrittura e consente di bloccare zone del file con accesso esclusivo, come vedrete in seguito.

Una tecnica per immettere byte in un **ByteBuffer** consiste nell'"introdurli" direttamente servendosi di uno dei metodi "put" per l'inserimento di uno o più byte o di valori di tipi primitivi. Tuttavia, come avete visto nell'esempio precedente, potete anche "inglobare" un array **byte** in un **ByteBuffer** ricorrendo al metodo **wrap()**; in questo caso l'array in questione non viene copiato, ma utilizzato come area di memorizzazione per il **ByteBuffer** generato: si dice infatti che il **ByteBuffer** è "supportato" dall'array.

Il file **data.txt** viene riaperto servendosi di un **RandomAccessFile**. È possibile spostare **FileChannel** all'interno del file; in questo caso specifico è spostato alla fine del file, in modo da potervi accodare altro testo.

Per l'accesso in sola lettura dovete assegnare esplicitamente un **ByteBuffer** utilizzando il metodo **static allocate()**. Come si è detto, l'obiettivo di **nio** è spostare velocemente grandi quantità di dati, pertanto le dimensioni di **ByteBuffer** dovrebbero essere adeguate. In realtà, la dimensione di **1 kilobyte** utilizzata nell'esempio è probabilmente insufficiente per un impiego normale: dovreste eseguire alcune prove con la vostra applicazione per trovare le dimensioni appropriate.

È anche possibile migliorare ulteriormente le prestazioni di velocità servendosi di **allocateDirect()** invece che **allocate()**, per produrre un buffer "diretto" il quale potrebbe integrarsi meglio con il sistema operativo utilizzato. Tuttavia, l'onere di tale allocazione è di gran lunga superiore e l'implementazione effettiva varia in funzione del sistema operativo: anche in questo caso, dovrete sperimentare con la vostra applicazione per scoprire se i buffer diretti vi offriranno vantaggi in termini di velocità.

Dopo avere chiamato **read()** per ordinare a **FileChannel** di memorizzare i byte in **ByteBuffer**, dovete chiamare il metodo **flip()** sul buffer per prepararlo all'estrazione dei byte: effettivamente questa tecnica è piuttosto elementare, ma ricordate che è di basso livello e ottimizzata per la massima velocità. Analogamente, quando utilizzerete il buffer per altre sessioni di lettura con

**read()**, dovrete chiamare **clear()** per prepararlo a ogni lettura. Il meccanismo è illustrato in questo semplice programma di copia:

```
//: io/ChannelCopy.java
// Copia di un file mediante canali e buffer
// {Args: ChannelCopy.java test.txt}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class ChannelCopy {
  private static final int BSIZE = 1024;
  public static void main(String[] args) throws Exception {
    if(args.length != 2) {
      System.out.println("arguments: sourcefile destfile");
      System.exit(1);
    }
    FileChannel
      in = new FileInputStream(args[0]).getChannel(),
      out = new FileOutputStream(args[1]).getChannel();
    ByteBuffer buffer = ByteBuffer.allocate(BSIZE);
    while(in.read(buffer) != -1) {
      buffer.flip(); // Preparazione per scrittura
      out.write(buffer);
      buffer.clear();  // Preparazione per lettura
    }
  }
} ///:~
```

Notate che viene aperto un **FileChannel** per la lettura e uno per la scrittura, e viene allocato un **ByteBuffer**; quando **FileChannel.read()** restituisce **-1** (senza dubbio un vecchio residuo di UNIX e di C) significa che la fine dell'input è stata raggiunta. Dopo ogni **read()**, che inserisce i dati nel buffer, il metodo **flip()** prepara il buffer in modo da estrarne le informazioni per mezzo di **write()**. Dopo avere chiamato il metodo **write()** le informazioni risiederanno ancora nel buffer, pertanto dovrete ricorrere a **clear()** per reimpostare tutti i puntatori interni affinché siano pronti ad accettare i dati di un'altra operazione di **read()**.

Il programma precedente non è certo l'approccio ideale per gestire questo tipo di operazioni; i metodi speciali **transferTo()** e **transferFrom()** vi consentono di collegare direttamente i canali tra loro:

```
//: io/TransferTo.java
// Utilizzo di transferTo() tra canali
// {Args: TransferTo.java TransferTo.txt}
import java.nio.channels.*;
import java.io.*;

public class TransferTo {
  public static void main(String[] args) throws Exception {
    if(args.length != 2) {
      System.out.println("arguments: sourcefile destfile");
      System.exit(1);
    }
    FileChannel
      in = new FileInputStream(args[0]).getChannel(),
      out = new FileOutputStream(args[1]).getChannel();
    in.transferTo(0, in.size(), out);
    // Oppure:
    // out.transferFrom(in, 0, in.size());
  }
} ///:~
```

Tenete presente che questa tecnica non è di utilizzo quotidiano, ma in ogni caso è utile conoscerla.

## *Conversione dei dati*

Riesaminando **GetChannel.java**, noterete che per visualizzare le informazioni contenute nel file i dati vengono estratti un **byte** alla volta, e ogni **byte** viene sottoposto a cast a un **char**. Questo meccanismo sembra alquanto primitivo, considerato che la classe **java.nio.CharBuffer** dispone di un metodo **toString()** che restituisce una stringa contenente i caratteri presenti nel buffer corrente.

Dal momento che un **ByteBuffer** può essere considerato come un **CharBuffer** con il metodo **asCharBuffer()**, perché non servirsene? Come potete vedere

dalla prima riga di output del programma seguente, questo approccio non funziona:

```
//: io/BufferToText.java
// Conversione di testo verso/da ByteBuffer
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.io.*;

public class BufferToText {
  private static final int BSIZE = 1024;
  public static void main(String[] args) throws Exception {
    FileChannel fc =
      new FileOutputStream("data2.txt").getChannel();
    fc.write(ByteBuffer.wrap("Some text".getBytes()));
    fc.close();
    fc = new FileInputStream("data2.txt").getChannel();
    ByteBuffer buff = ByteBuffer.allocate(BSIZE);
    fc.read(buff);
    buff.flip();
    // Non funziona:
    System.out.println(buff.asCharBuffer());
    // Decodifica mediante il Charset predefinito per il
    // computer corrente:
    buff.rewind();
    String encoding = System.getProperty("file.encoding");
    System.out.println("Decoded using " + encoding + ": "
      + Charset.forName(encoding).decode(buff));
    // In alternativa, potreste codificare in un formato
    // visualizzabile:
    fc = new FileOutputStream("data2.txt").getChannel();
    fc.write(ByteBuffer.wrap(
      "Some text".getBytes("UTF-16BE")));
    fc.close();
    // Nuovo tentativo di lettura:
```

```
      fc = new FileInputStream("data2.txt").getChannel();
      buff.clear();
      fc.read(buff);
      buff.flip();
      System.out.println(buff.asCharBuffer());
      // Utilizzo di un CharBuffer per la scrittura
      fc = new FileOutputStream("data2.txt").getChannel();
      buff = ByteBuffer.allocate(24); // Piu' di quanto richiesto
      buff.asCharBuffer().put("Some text");
      fc.write(buff);
      fc.close();
      // Lettura e visualizzazione:
      fc = new FileInputStream("data2.txt").getChannel();
      buff.clear();
      fc.read(buff);
      buff.flip();
      System.out.println(buff.asCharBuffer());
    }
  } /* Output:
  ????
  Decoded using Cp1252: Some text
  Some text
  Some text
  *///:~
```

Il buffer contiene normali byte; per trasformarli in caratteri occorre *codificarli* in fase d'inserimento, in modo che abbiano senso quando estratti, oppure *decodificarli* in fase di estrazione dal buffer. Queste operazioni possono essere svolte ricorrendo alla classe **java.nio.charset.Charset**, che fornisce gli strumenti per la codifica in diversi set di caratteri:

```
//: io/AvailableCharSets.java
// Visualizzazione di Charsets e alias
import java.nio.charset.*;
import java.util.*;
import static net.mindview.util.Print.*;
```

```
public class AvailableCharSets {
  public static void main(String[] args) {
    SortedMap<String,Charset> charSets =
      Charset.availableCharsets();
    Iterator<String> it = charSets.keySet().iterator();
    while(it.hasNext()) {
      String csName = it.next();
      printnb(csName);
      Iterator aliases =
        charSets.get(csName).aliases().iterator();
      if(aliases.hasNext())
        printnb(": ");
      while(aliases.hasNext()) {
        printnb(aliases.next());
        if(aliases.hasNext())
          printnb(", ");
      }
      print();
    }
  }
} /* Output:
Big5: csBig5
Big5-HKSCS: big5-hkscs, big5hk, big5-hkscs:unicode3.0,
big5hkscs, Big5_HKSCS
EUC-JP: eucjis, x-eucjp, csEUCPkdFmtjapanese, eucjp, Extended_
UNIX_Code_Packed_Format_for_Japanese, x-euc-jp, euc_jp
EUC-KR: ksc5601, 5601, ksc5601_1987, ksc_5601, ksc5601-1987,
euc_kr, ks_c_5601-1987, euckr, csEUCKR
GB18030: gb18030-2000
GB2312: gb2312-1980, gb2312, EUC_CN, gb2312-80, euc-cn, euccn,
x-EUC-CN
GBK: windows-936, CP936
...
*///:~
```

Quindi, per tornare all'analisi di **BufferToText.java**, se eseguite il metodo **rewind()** sul buffer (per ritornare all'inizio dei dati) e utilizzate il set di caratteri predefinito per la piattaforma corrente con il metodo **decode()** (per deco-

dificare i dati), otterrete la visualizzazione corretta del **CharBuffer** risultante. Per determinare qual è il set di caratteri predefinito, servitevi innanzitutto dell'istruzione **System.getProperty("file.encoding")** per produrre una stringa con il nome del set; passando tale nome a **Charset.forName()** otterrete l'oggetto **Charset**, che potrete utilizzare per decodificare la stringa.

Un'alternativa è la codifica mediante **encode()** in un set di caratteri che non dia problemi di visualizzazione in fase di lettura del file, come avete visto nella terza parte di **BufferToText.java**. In questo esempio specifico è stato utilizzato il set UTF-16BE per scrivere il testo nel file: al momento della lettura, è bastato convertirlo in un **CharBuffer** per ottenere il testo previsto.

Per concludere, avete visto che cosa accade quando *scrivete* nel **ByteBuffer** con un **CharBuffer**: affronteremo meglio l'argomento nel prosieguo del capitolo. Notate che a **ByteBuffer** sono stati assegnati 24 byte; dal momento che ogni **char** richiede due byte, queste dimensioni sono sufficienti per 12 **char**, ma "Some text" ne contiene soltanto 9. Come potete notare dall'output, gli zero byte restanti (e basta) compaiono nella rappresentazione del **CharBuffer** prodotta dal metodo **toString()**.

> **Esercizio 23** (6) Create e testate un metodo di utilità per visualizzare il contenuto di un **CharBuffer** fino al punto in cui i caratteri non siano più visualizzabili.

## *Ottenimento di tipi di dato primitivi*

Benché un **ByteBuffer** contenga soltanto alcuni byte, offre dei metodi per produrre i diversi tipi di dato dai byte stessi. Questo esempio si riferisce all'inserimento e all'estrazione di diversi valori, per mezzo di tali metodi:

```
//: io/GetData.java
// Ottenimento di diverse rappresentazioni da un ByteBuffer
import java.nio.*;
import static net.mindview.util.Print.*;

public class GetData {
  private static final int BSIZE = 1024;
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.allocate(BSIZE);
    // L'allocazione azzera automaticamente il ByteBuffer:
    int i = 0;
    while(i++ < bb.limit())
```

```
      if(bb.get() != 0)
        print("nonzero");
    print("i = " + i);
    bb.rewind();
    // Registra e legge un array di char:
    bb.asCharBuffer().put("Howdy!");
    char c;
    while((c = bb.getChar()) != 0)
      printnb(c + " ");
    print();
    bb.rewind();
    // Registra e legge uno short:
    bb.asShortBuffer().put((short)471142);
    print(bb.getShort());
    bb.rewind();
    // Registra e legge un int:
    bb.asIntBuffer().put(99471142);
    print(bb.getInt());
    bb.rewind();
    // Registra e legge un long:
    bb.asLongBuffer().put(99471142);
    print(bb.getLong());
    bb.rewind();
    // Registra e legge un float:
    bb.asFloatBuffer().put(99471142);
    print(bb.getFloat());
    bb.rewind();
    // Registra e legge un double:
    bb.asDoubleBuffer().put(99471142);
    print(bb.getDouble());
    bb.rewind();
  }
} /* Output:
i = 1025
H o w d y !
12390
99471142
```

```
99471142
9.9471144E7
9.9471142E7
*///:~
```

Dopo l'allocazione di un **ByteBuffer**, i relativi valori vengono controllati per verificare se l'allocazione del buffer azzera automaticamente il contenuto, cosa che in effetti avviene. Tutti i 1024 valori vengono verificati (fino al valore **limit()** del buffer) e tutti sono pari a zero.

Il modo più semplice per inserire valori primitivi in un **ByteBuffer** è ottenere la "vista" adatta sul buffer in questione, ricorrendo ai metodi **asCharBuffer()**, **asShortBuffer()** ecc., quindi utilizzare il metodo **put()** della vista. Come potete vedere, questa è la tecnica adottata per ciascuno dei tipi di dati primitivi. L'unico metodo **put()** che differisce dagli altri è quello per **ShortBuffer**, che richiede una conversione: notate che il cast tronca e modifica il valore risultante.

## *Buffer per la visualizzazione*

Un "buffer di vista" (*view buffer*) vi consente di operare con il **ByteBuffer** sottostante attraverso l'"ottica" di un determinato tipo primitivo; tuttavia, il **ByteBuffer** continua a essere lo strumento di memorizzazione effettivo che "supporta" la vista, in modo che tutti i cambiamenti eseguiti sulla vista si rispecchino in altrettante modifiche ai dati nel **ByteBuffer**. Come avete visto nell'esempio precedente, questo meccanismo consente di inserire in modo pratico i tipi primitivi in un **ByteBuffer**. Una vista consente anche di leggere i valori primitivi da un **ByteBuffer**, uno per volta (come previsto da **ByteBuffer**) o in gruppo (negli array). Ecco un esempio che manipola gli **int** in un **ByteBuffer** tramite un **IntBuffer**:

```
//: io/IntBufferDemo.java
// Manipolazione dei dati int in un ByteBuffer, mediante
// IntBuffer
import java.nio.*;

public class IntBufferDemo {
  private static final int BSIZE = 1024;
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.allocate(BSIZE);
    IntBuffer ib = bb.asIntBuffer();
    // Registra un array di int:
```

```
    ib.put(new int[]{ 11, 42, 47, 99, 143, 811, 1016 });
    // Lettura e scrittura in posizione assoluta:
    System.out.println(ib.get(3));
    ib.put(3, 1811);
    // Impostazione di un nuovo limite prima di riavvolgere
    // il buffer.
    ib.flip();
    while(ib.hasRemaining()) {
      int i = ib.get();
      System.out.println(i);
    }
  }
} /* Output:
99
11
42
47
1811
143
811
1016
*///:~
```

Il metodo **put()** sovraccarico viene inizialmente utilizzato per memorizzare un array di **int**. Le successive chiamate ai metodi **get()** e **put()** accedono direttamente a una posizione **int** nel **ByteBuffer** sottostante. Ricordate che questi accessi a posizioni assolute sono disponibili per i tipi primitivi anche comunicando con il **ByteBuffer** in modo diretto.

Non appena il **ByteBuffer** sottostante sarà stato popolato di **int** o di altri tipi primitivi tramite un buffer di vista, sarà possibile scrivere il **ByteBuffer** su un canale; fatto questo, non vi sarà difficile leggere da quel canale e utilizzare un buffer di vista per eseguire la conversione nel tipo di primitivo specifico. L'esempio seguente interpreta la stessa sequenza di byte come **short**, **int**, **float**, **long** e **double**, producendo diversi buffer di vista sullo stesso **ByteBuffer**:

```
//: io/ViewBuffers.java
import java.nio.*;
import static net.mindview.util.Print.*;
```

```
public class ViewBuffers {
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.wrap(
      new byte[]{ 0, 0, 0, 0, 0, 0, 0, 'a' });
    bb.rewind();
    printnb("Byte Buffer ");
    while(bb.hasRemaining())
      printnb(bb.position()+ " -> " + bb.get() + ", ");
    print();
    CharBuffer cb =
      ((ByteBuffer)bb.rewind()).asCharBuffer();
    printnb("Char Buffer ");
    while(cb.hasRemaining())
      printnb(cb.position() + " -> " + cb.get() + ", ");
    print();
    FloatBuffer fb =
      ((ByteBuffer)bb.rewind()).asFloatBuffer();
    printnb("Float Buffer ");
    while(fb.hasRemaining())
      printnb(fb.position()+ " -> " + fb.get() + ", ");
    print();
    IntBuffer ib =
      ((ByteBuffer)bb.rewind()).asIntBuffer();
    printnb("Int Buffer ");
    while(ib.hasRemaining())
      printnb(ib.position()+ " -> " + ib.get() + ", ");
    print();
    LongBuffer lb =
      ((ByteBuffer)bb.rewind()).asLongBuffer();
    printnb("Long Buffer ");
    while(lb.hasRemaining())
      printnb(lb.position()+ " -> " + lb.get() + ", ");
    print();
    ShortBuffer sb =
      ((ByteBuffer)bb.rewind()).asShortBuffer();
    printnb("Short Buffer ");
```

```
    while(sb.hasRemaining())
      printnb(sb.position()+ " -> " + sb.get() + ", ");
    print();
    DoubleBuffer db =
      ((ByteBuffer)bb.rewind()).asDoubleBuffer();
    printnb("Double Buffer ");
    while(db.hasRemaining())
      printnb(db.position()+ " -> " + db.get() + ", ");
  }
} /* Output:
Byte Buffer 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 0, 4 -> 0, 5 -> 0,
6 -> 0, 7 -> 97,
Char Buffer 0 ->  , 1 ->  , 2 ->  , 3 -> a,
Float Buffer 0 -> 0.0, 1 -> 1.36E-43,
Int Buffer 0 -> 0, 1 -> 97,
Long Buffer 0 -> 97,
Short Buffer 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 97,
Double Buffer 0 -> 4.8E-322,
*///:~
```

Il **ByteBuffer** viene prodotto "incorporando" un array di otto **byte**, poi visualizzato tramite i buffer di vista di tutti i tipi primitivi. Nel seguente schema, potete vedere come i dati vengono rappresentati diversamente in lettura dai vari tipi di buffer.

<table>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>97</td><td>byte</td></tr>
<tr><td colspan="2"></td><td colspan="2"></td><td colspan="2"></td><td colspan="2">a</td><td>char</td></tr>
<tr><td colspan="2">0</td><td colspan="2">0</td><td colspan="2">0</td><td colspan="2">97</td><td>short</td></tr>
<tr><td colspan="4">0</td><td colspan="4">97</td><td>int</td></tr>
<tr><td colspan="4">0.0</td><td colspan="4">1.36E-43</td><td>float</td></tr>
<tr><td colspan="8">97</td><td>long</td></tr>
<tr><td colspan="8">4.8E-322</td><td>double</td></tr>
</table>

Questo diagramma corrisponde all'output del programma.

**Esercizio 24** (1) Modificate **IntBufferDemo.java** per utilizzare i **double**.

### Big endian e little endian

Computer diversi potrebbero utilizzare metodi di ordinamento diversi per memorizzare i dati: secondo l'approccio "big endian", il byte più significativo viene collocato nell'indirizzo di memoria più basso, mentre con il "little endian" il byte più significativo si trova nell'indirizzo di memoria più elevato. Quando memorizzate una quantità di **byte** superiore a uno, come avviene per gli **int**, i **float** ecc., potreste avere bisogno di considerare questo ordinamento dei byte.

Un **ByteBuffer** memorizza i dati nella forma big endian, come pure avviene per i dati trasmessi tramite rete. Potete modificare il tipo di ordinamento di un **ByteBuffer** ricorrendo al metodo **order()** con gli argomenti **ByteOrder.BIG_ENDIAN** oppure **ByteOrder.LITTLE_ENDIAN**.

La differenza tra i due ordinamenti può essere notevole. Considerate, per esempio, un **ByteBuffer** che contiene i seguenti due byte:

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| b1 | | | | | | | | b2 | | | | | | | |

Se leggete i dati sotto forma di **short** (**ByteBuffer.asShortBuffer()**) otterrete il numero 97 (00000000 01100001), ma se cambiate in little endian il numero sarà 24.832 (01100001 00000000). Ecco un esempio che mostra come varia l'ordinamento dei byte nei caratteri in base alle impostazioni endian:

```
//: io/Endians.java
// Differenze di ordinamento endian e memorizzazione dei dati.
import java.nio.*;
import java.util.*;
import static net.mindview.util.Print.*;

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

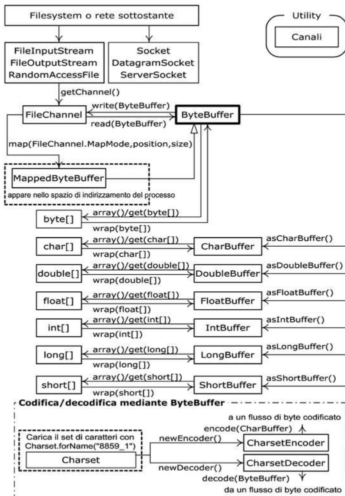
```
        ByteBuffer bb = ByteBuffer.wrap(new byte[12]);
        bb.asCharBuffer().put("abcdef");
        print(Arrays.toString(bb.array()));
        bb.rewind();
        bb.order(ByteOrder.BIG_ENDIAN);
        bb.asCharBuffer().put("abcdef");
        print(Arrays.toString(bb.array()));
        bb.rewind();
        bb.order(ByteOrder.LITTLE_ENDIAN);
        bb.asCharBuffer().put("abcdef");
        print(Arrays.toString(bb.array()));
    }
} /* Output:
[0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]
[0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]
[97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102, 0]
*///:~
```

Al **ByteBuffer** è dato abbastanza spazio per contenere tutti i byte di un array di char come buffer esterno, in modo da chiamare il metodo **array()** per visualizzare i byte sottostanti. Il metodo **array()** è "facoltativo", e può essere chiamato soltanto a fronte di un buffer supportato da un array: in caso contrario, si ottiene un'eccezione di tipo **UnsupportedOperationException**.

L'array di char è inserito in **ByteBuffer** per mezzo di una vista di **CharBuffer**. Quando i byte sottostanti vengono visualizzati, notate che l'ordinamento predefinito corrisponde al successivo ordinamento big endian, mentre con little endian si ottiene la permutazione dei byte.

## *Manipolazione dei dati tramite buffer*

Lo schema a pagina seguente illustra le relazioni tra le classi **nio**, affinché possiate avere ben chiaro come spostare e convertire i dati. Per esempio, se volete scrivere un array di **byte** in un file dovrete inglobare l'array di **byte** utilizzando il metodo **ByteBuffer.wrap()**, aprire un canale su **FileOutput-Stream** con il metodo **getChannel()** e scrivere i dati in **FileChannel** da questo **ByteBuffer**.




Tenete presente che **ByteBuffer** è il solo modo per spostare i dati all'interno e dall'interno dei canali, e che è possibile creare soltanto un buffer autonomo di tipi primitivi, oppure ottenerne uno utilizzando **ByteBuffer** come metodo

"as", come in **asCharBuffer()**. In altri termini, non potete convertire un buffer autonomo di tipi primitivi *in* un **ByteBuffer**. Tuttavia, considerato che è consentito spostare i dati dentro e fuori da un **ByteBuffer** mediante un buffer di vista, questo non costituisce un vero e proprio limite.

## *I buffer in dettaglio*

Un **Buffer** consiste di dati e di quattro indici che consentono di operare con questi dati in modo efficiente: *mark* (contrassegno), *position* (posizione), *limit* (limite) e *capacità* (capienza). Esistono metodi per impostare e ripristinare questi indici e per interrogarne il valore.

| | |
|---|---|
| **capacity( )** | Restituisce la capienza del buffer. |
| **clear( )** | Svuota il buffer, imposta la posizione al valore zero e il limite a quello della capienza. Utilizzate questo metodo per sovrascrivere un buffer esistente. |
| **flip( )** | Imposta il limite al valore della posizione, e la posizione a zero. Questo metodo è utilizzato per preparare il buffer alla lettura dopo che vi sono stati scritti alcuni dati. |
| **limit( )** | Restituisce il valore del limite. |
| **limit(int lim)** | Imposta il valore del limite. |
| **mark( )** | Imposta il contrassegno al valore della posizione. |
| **position( )** | Restituisce il valore della posizione. |
| **position(int pos)** | Imposta il valore della posizione. |
| **remaining( )** | Restituisce un valore corrispondente alla differenza tra limite e posizione. |
| **hasRemaining( )** | Restituisce **true** se tra la posizione e il limite esistono elementi. |

I metodi che inseriscono ed estraggono i dati dal buffer aggiornano questi indici a seguito dei cambiamenti intervenuti.

Questo esempio utilizza un algoritmo molto semplice che scambia la posizione di due caratteri adiacenti, per mescolare i caratteri di un **CharBuffer**:

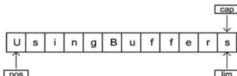
```
//: io/UsingBuffers.java
import java.nio.*;
import static net.mindview.util.Print.*;

public class UsingBuffers {
```

```
  private static void symmetricScramble(CharBuffer buffer){
    while(buffer.hasRemaining()) {
      buffer.mark();
      char c1 = buffer.get();
      char c2 = buffer.get();
      buffer.reset();
      buffer.put(c2).put(c1);
    }
  }
  public static void main(String[] args) {
    char[] data = "UsingBuffers".toCharArray();
    ByteBuffer bb = ByteBuffer.allocate(data.length * 2);
    CharBuffer cb = bb.asCharBuffer();
    cb.put(data);
    print(cb.rewind());
    symmetricScramble(cb);
    print(cb.rewind());
    symmetricScramble(cb);
    print(cb.rewind());
  }
} /* Output:
UsingBuffers
sUniBgfuefsr
UsingBuffers
*///:~
```

Anche se potreste generare direttamente un **CharBuffer** chiamando **wrap()** su un array di **char**, al suo posto viene allocato un **ByteBuffer** di supporto: il **CharBuffer** viene prodotto come vista su **ByteBuffer**. Questa tecnica evidenzia il fatto che l'obiettivo è sempre quello di manipolare un **ByteBuffer**, dal momento che questo consente di interagire con un canale.

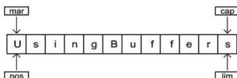
Osservate la rappresentazione del buffer all'entrata del metodo **symmetricScramble()** mostrato di seguito.



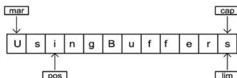
La posizione (*pos*) punta al primo elemento nel buffer, mentre la capienza (*cap*) e il limite (*lim*) fanno riferimento all'ultimo elemento.

In **symmetricScramble()**, il ciclo **while** itera fino a quando sono presenti ulteriori elementi nel buffer. La posizione del buffer cambia ogni volta che viene chiamata una funzione **get()** o **put()** su di esso. È possibile chiamare tali metodi in maniera "assoluta", cioè indicando come argomento l'indice sul quale **get()** e **put()** devono essere applicate: in questo modo non viene modificato il valore della posizione del buffer.

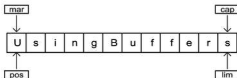
Quando il controllo entra nel ciclo **while**, il valore del contrassegno è impostato per mezzo di una chiamata a **mark()**. La condizione del buffer diventa quindi:



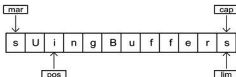

Le due chiamate **get()** relative salvano i valori dei primi due caratteri nelle variabili **c1** e **c2**. Dopo queste due chiamate, il buffer avrà l'aspetto seguente:



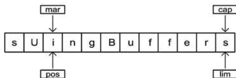

Per eseguire lo scambio, occorre scrivere **c2** in *position* = 0 e **c1** in *position* = 1. A questo scopo, potete servirvi del metodo **put()** assoluto oppure impostare la posizione al valore del contrassegno, che è esattamente l'operazione svolta da **reset()**:



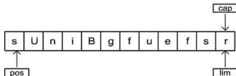
I due metodi **put()** scrivono prima **c2** e poi **c1**:




In occasione della successiva ripetizione del ciclo, il contrassegno viene impostato al valore corrente della posizione:




Questo processo continua fino all'esaurimento del buffer; al termine del ciclo **while**, la posizione si trova alla fine del buffer. Se visualizzate il buffer l'output conterrà soltanto i caratteri tra la posizione e il limite. Pertanto, per visualizzare l'intero contenuto del buffer dovete impostare la posizione all'inizio del buffer con **rewind()**. Questo è lo stato del buffer dopo la chiamata a **rewind()**; tenete presente che il valore del *contrassegno* diventa indefinito:




Alla successiva chiamata della funzione **symmetricScramble()**, il **CharBuffer** subisce lo stesso processo e viene ripristinato nella sua condizione originale.

## File mappati in memoria

I file "mappati" in memoria (*memory-mapped*) consentono di creare e modificare file che sono troppo grandi per essere contenuti in memoria. Servendovi di un file di tipo memory mapped, potete considerare che l'intero file è ospitato in memoria e accedervi come fareste con un array di grandi dimensioni. Questo approccio semplifica in modo notevole il codice da scrivere per gestire il file. Di seguito è mostrato un breve esempio di questa tecnica:

```
//: io/LargeMappedFiles.java
// Creazione di file di grandi dimensioni utilizzando
// la tecnica memory mapping.
// {RunByHand}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;
import static net.mindview.util.Print.*;

public class LargeMappedFiles {
  static int length = 0x8FFFFFF; // 128 MB
  public static void main(String[] args) throws Exception {
    MappedByteBuffer out =
      new RandomAccessFile("test.dat", "rw").getChannel()
      .map(FileChannel.MapMode.READ_WRITE, 0, length);
    for(int i = 0; i < length; i++)
      out.put((byte)'x');
    print("Finished writing");
    for(int i = length/2; i < length/2 + 6; i++)
      printnb((char)out.get(i));
  }
} ///:~
```

Per lavorare sia in scrittura sia in lettura, iniziate con un **RandomAccessFile** al fine di ottenere un canale per il file, quindi chiamate il metodo **map()** per creare un **MappedByteBuffer**, che è un tipo particolare di buffer diretto. Ricordate che dovete specificare il punto di partenza e le dimensioni della porzione di file da sottoporre a mappatura: questo significa che potete lavorare con aree ridotte di un file molto grande.

La classe **MappedByteBuffer** è ereditata da **ByteBuffer**, di conseguenza ne possiede tutti i metodi. Sebbene questo esempio mostri unicamente impieghi molto semplici dei metodi **put()** e **get()**, tenete presente che sono disponibili anche metodi quali **asCharBuffer()**.

Il file creato dal programma precedente è di 128 MB, una dimensione probabilmente maggiore di quanto il vostro sistema operativo consenta di collocare in memoria. Il file sembra essere interamente accessibile poiché soltanto alcune sue porzioni sono presenti in memoria, mentre il resto viene gestito come swap; in questo modo, potete modificare senza problemi un file fino a 2 GB. Tenete presente che, per ottimizzare le prestazioni, questo meccanismo utilizza le funzionalità di file mapping del sistema operativo.

### Prestazioni

Nonostante le prestazioni della "vecchia" libreria di flussi I/O siano già migliorate in seguito all'implementazione di **nio**, l'accesso ai file mappati si rivela ancora più rapido. Questo programma esegue un semplice confronto di prestazioni:

```
//: io/MappedIO.java
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class MappedIO {
  private static int numOfInts = 4000000;
  private static int numOfUbuffInts = 200000;
  private abstract static class Tester {
    private String name;
    public Tester(String name) { this.name = name; }
    public void runTest() {
      System.out.print(name + ": ");
      try {
        long start = System.nanoTime();
        test();
        double duration = System.nanoTime() - start;
        System.out.format("%.2f\n", duration/1.0e9);
      } catch(IOException e) {
        throw new RuntimeException(e);
```

```
      }
    }
    public abstract void test() throws IOException;
  }
  private static Tester[] tests = {
    new Tester("Stream  Write") {
      public void test() throws IOException {
        DataOutputStream dos = new DataOutputStream(
          new BufferedOutputStream(
            new FileOutputStream(new File("temp.tmp"))));
        for(int i = 0; i < numOfInts; i++)
          dos.writeInt(i);
        dos.close();
      }
    },
    new Tester("Mapped Write") {
      public void test() throws IOException {
        FileChannel fc =
          new RandomAccessFile("temp.tmp", "rw")
          .getChannel();
        IntBuffer ib = fc.map(
          FileChannel.MapMode.READ_WRITE, 0, fc.size())
          .asIntBuffer();
        for(int i = 0; i < numOfInts; i++)
          ib.put(i);
        fc.close();
      }
    },
    new Tester("Stream Read") {
      public void test() throws IOException {
        DataInputStream dis = new DataInputStream(
          new BufferedInputStream(
            new FileInputStream("temp.tmp")));
        for(int i = 0; i < numOfInts; i++)
          dis.readInt();
        dis.close();
      }
```

```
},
new Tester("Mapped Read") {
  public void test() throws IOException {
    FileChannel fc = new FileInputStream(
      new File("temp.tmp")).getChannel();
    IntBuffer ib = fc.map(
      FileChannel.MapMode.READ_ONLY, 0, fc.size())
      .asIntBuffer();
    while(ib.hasRemaining())
      ib.get();
    fc.close();
  }
},
new Tester("Stream Read/Write") {
  public void test() throws IOException {
    RandomAccessFile raf = new RandomAccessFile(
      new File("temp.tmp"), "rw");
    raf.writeInt(1);
    for(int i = 0; i < numOfUbuffInts; i++) {
      raf.seek(raf.length() - 4);
      raf.writeInt(raf.readInt());
    }
    raf.close();
  }
},
new Tester("Mapped Read/Write") {
  public void test() throws IOException {
    FileChannel fc = new RandomAccessFile(
      new File("temp.tmp"), "rw").getChannel();
    IntBuffer ib = fc.map(
      FileChannel.MapMode.READ_WRITE, 0, fc.size())
      .asIntBuffer();
    ib.put(0);
    for(int i = 1; i < numOfUbuffInts; i++)
      ib.put(ib.get(i - 1));
    fc.close();
```

```
      }
    }
  };
  public static void main(String[] args) {
    for(Tester test : tests)
      test.runTest();
  }
} /* Output: (90% match)
Stream Write: 0.56
Mapped Write: 0.12
Stream Read: 0.80
Mapped Read: 0.07
Stream Read/Write: 5.32
Mapped Read/Write: 0.02
*///:~
```

Come avete visto in precedenza in alcuni esempi, il metodo **runTest()** viene utilizzato dal design pattern *Template Method* per creare un ambiente di test per varie implementazioni del metodo **test()** definite in classi interne anonime. Ciascuna di queste sottoclassi esegue un tipo di test, cosicché i metodi **test()** forniscono anche un prototipo per l'esecuzione di varie attività di I/O.

Anche se un'operazione di scrittura di tipo mapped sembra utilizzare un **FileOutputStream**, tutto l'output deve utilizzare un **RandomAccessFile**, come mostra la sezione lettura/scrittura del codice precedente.

Tenete presente che i metodi **test()** includono il tempo necessario per inizializzare i vari oggetti di I/O. Dai risultati si evince che, sebbene l'impostazione necessaria per utilizzare i file mapped sia un'attività onerosa, il vantaggio in termini di prestazioni confrontato con i flussi di I/O è comunque molto elevato.

**Esercizio 25** (6) Utilizzando gli esempi di questo capitolo, eseguite alcune prove modificando le istruzioni **ByteBuffer.allocate()** in **ByteBuffer.allocateDirect()**. Dimostrate le differenze nelle prestazioni, senza trascurare le modifiche nei tempi di avvio dei programmi.

**Esercizio 26** (3) Modificate **strings/JGrep.java** per utilizzare i file mappati in memoria **nio**.

## *Blocco dei file*

Il blocco dei file consente di sincronizzare l'accesso a un file come risorsa condivisa. Nulla vieta, però, che due thread che si contendono lo stesso file possano risiedere in JVM diverse, oppure uno potrebbe essere un thread Java e l'altro un thread nativo del sistema operativo. Per questo motivo, il blocco sui file è visibile agli altri processi del sistema operativo: infatti, la tecnologia di bloccaggio dei file utilizzata da Java ricorre direttamente alle funzioni di blocco del sistema operativo.

Considerate questo semplice esempio di blocco di file.

```
//: io/FileLocking.java
import java.nio.channels.*;
import java.util.concurrent.*;
import java.io.*;

public class FileLocking {
  public static void main(String[] args) throws Exception {
    FileOutputStream fos= new FileOutputStream("file.txt");
    FileLock fl = fos.getChannel().tryLock();
    if(fl != null) {
      System.out.println("Locked File");
      TimeUnit.MILLISECONDS.sleep(100);
      fl.release();
      System.out.println("Released Lock");
    }
    fos.close();
  }
} /* Output:
Locked File
Released Lock
*///:~
```

Chiamando i metodi **tryLock()** o **lock()** su un **FileChannel**, ottenete un **FileLock** sull'intero file. **SocketChannel**, **DatagramChannel** e **ServerSocketChannel** non richiedono blocchi, poiché sono intrinsecamente entità a processo singolo: di norma infatti un socket di rete non viene ripartito tra due processi. Il metodo **tryLock()** non è bloccante: cerca di impostare il

blocco, ma se questo non è possibile, magari perché un altro processo detiene già lo stesso blocco e non è di tipo condiviso, ritorna semplicemente dalla chiamata di metodo. Il metodo **lock()**, invece, è bloccante e rimane tale fino a quando non riesce ad acquisire il blocco sul file, finché il thread che ha invocato il metodo **lock()** viene interrotto, o ancora fino a quando il canale su cui è stato chiamato **lock()** viene chiuso. Un blocco può essere rilasciato chiamando **FileLock.release()**.

È anche possibile bloccare soltanto una parte del file, con:

```
tryLock(long position, long size, boolean shared)
```

oppure

```
lock(long position, long size, boolean shared)
```

che bloccano una determinata porzione del file, corrispondente a (**size - position**). Il terzo argomento indica se il blocco è condiviso.

Anche se i metodi di blocco privi di argomenti si adattano alle modifiche delle dimensioni di un file, i blocchi a dimensione fissa non cambiano con il variare delle dimensioni del file. Se viene acquisito un blocco per un'area che va da **position** a **position + size** e la dimensione del file aumenta oltre **position + size**, la sezione che si trova oltre **position + size** non viene bloccata. I metodi di blocco privi di argomenti mantengono il blocco sull'intero file, anche se le sue dimensioni aumentano.

Il supporto per i blocchi esclusivi (*exclusive lock*) e condivisi (*shared lock*) deve essere fornito dal sistema operativo. Qualora il sistema operativo non supporti il blocco condiviso e ne venga richiesto uno, come alternativa verrà utilizzato di preferenza un blocco esclusivo. È possibile sapere se il blocco è condiviso o esclusivo interrogando il metodo **FileLock.isShared()**.

## Blocco parziale di file mappati

Come accennato in precedenza, il file mapping è in genere utilizzato in caso di file molto grandi. Potreste avere necessità di impostare un blocco su parti di file così voluminosi, in modo che altri processi possano modificare parti non bloccate dello stesso file. Questo è quanto accade, per esempio, con un database, che quindi è in grado di servire numerosi utenti simultaneamente.

Ecco un esempio con due thread, ciascuno dei quali blocca una parte diversa di un file:

```
//: io/LockingMappedFiles.java
// Blocco di porzioni di un file mapped.
// {RunByHand}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class LockingMappedFiles {
  static final int LENGTH = 0x8FFFFFF; // 128 MB
  static FileChannel fc;
  public static void main(String[] args) throws Exception {
    fc =
      new RandomAccessFile("test.dat", "rw").getChannel();
    MappedByteBuffer out =
      fc.map(FileChannel.MapMode.READ_WRITE, 0, LENGTH);
    for(int i = 0; i < LENGTH; i++)
      out.put((byte)'x');
    new LockAndModify(out, 0, 0 + LENGTH/3);
    new LockAndModify(out, LENGTH/2, LENGTH/2 + LENGTH/4);
  }
  private static class LockAndModify extends Thread {
    private ByteBuffer buff;
    private int start, end;
    LockAndModify(ByteBuffer mbb, int start, int end) {
      this.start = start;
      this.end = end;
      mbb.limit(end);
      mbb.position(start);
      buff = mbb.slice();
      start();
    }
    public void run() {
      try {
```

```
          // Lock esclusivo senza sovrapposizione:
          FileLock fl = fc.lock(start, end, false);
          System.out.println("Locked:  "+ start +" to "+ end);
          // Esegue modifiche:
          while(buff.position() < buff.limit() - 1)
            buff.put((byte)(buff.get() + 1));
          fl.release();
          System.out.println("Rileased: "+start+" to "+ end);
        } catch(IOException e) {
          throw new RuntimeException(e);
        }
      }
    }
} ///:~
```

La classe di thread **LockAndModify** imposta l'area del buffer, e con **slice()** ne crea una porzione da modificare; il metodo **run()** acquisisce il blocco sul canale del file. Ricordate che non potete ottenere un blocco sul buffer, ma soltanto sul canale. La chiamata a **lock()** è molto simile all'impostazione di un blocco di threading su un oggetto: dopo questa chiamata, avete a disposizione una "sezione critica" con accesso esclusivo a questa porzione del file.[5]

I blocchi vengono rilasciati automaticamente all'uscita dalla JVM, oppure quando viene chiuso il canale sul quale è stato impostato il blocco; potete però anche chiamare esplicitamente **release()** sull'oggetto **FileLock**, come nell'esempio precedente.

## Compressione

La libreria di I/O Java contiene le classi di supporto alla lettura e scrittura dei flussi in formato compresso, all'interno delle quali vengono inglobate le altre classi di I/O per garantire le funzionalità di compressione.

Queste classi non derivano da **Writer** e **Reader**, ma appartengono alle gerarchie di **OutputStream** e di **InputStream**, in quanto la libreria di compressione funziona soltanto con i byte.

---

5. L'argomento dei thread sarà trattato in maggiore dettaglio nel Volume 3, Capitolo 1.

A volte, tuttavia, potreste essere costretti a mescolare i due tipi di flussi; a questo proposito, ricordate che potete utilizzare **InputStreamReader** e **OutputStreamWriter** per convertire facilmente un tipo in un altro.

| *Classe di compressione* | *Funzione* |
|---|---|
| **CheckedInputStream** | **GetCheckSum()** produce la *checksum* di controllo per qualsiasi **InputStream**, non soltanto per la decompressione. |
| **CheckedOutputStream** | **GetCheckSum()** produce la *checksum* di controllo per qualsiasi **OutputStream**, non soltanto per la compressione. |
| **DeflaterOutputStream** | Classe di base per le classi di compressione. |
| **ZipOutputStream** | Un **DeflaterOutputStream** che comprime i dati in formato Zip. |
| **GZIPOutputStream** | Un **DeflaterOutputStream** che comprime i dati in formato GZIP. |
| **InflaterInputStream** | Classe di base per le classi di decompressione. |
| **ZipInputStream** | Un **InflaterInputStream** il quale decomprime i dati che sono stati memorizzati in formato Zip. |
| **GZIPInputStream** | Un **InflaterInputStream** il quale decomprime i dati che sono stati memorizzati in formato GZIP. |

Malgrado esistano molti algoritmi di compressione, Zip e GZIP sono tra quelli utilizzati con maggiore frequenza; in ogni caso, potrete gestire senza problemi i vostri dati compressi ricorrendo alle numerose utility disponibili per lettura e scrittura in questi formati.

## Compressione semplice con GZIP

L'interfaccia GZIP è semplice e pertanto probabilmente più adatta quando avete un solo flusso di dati da comprimere, in luogo di un contenitore con dati di diverso tipo. Questo codice è un esempio che comprime un solo file:

```
//: io/GZIPcompress.java
// {Args: GZIPcompress.java}
import java.util.zip.*;
import java.io.*;

public class GZIPcompress {
```

```
  public static void main(String[] args)
  throws IOException {
    if(args.length == 0) {
      System.out.println(
        "Usage: \nGZIPcompress file\n" +
        "\tUses GZIP compression to compress " +
        "the file to test.gz");
      System.exit(1);
    }
    BufferedReader in = new BufferedReader(
      new FileReader(args[0]));
    BufferedOutputStream out = new BufferedOutputStream(
      new GZIPOutputStream(
        new FileOutputStream("test.gz")));
    System.out.println("Writing file");
    int c;
    while((c = in.read()) != -1)
      out.write(c);
    in.close();
    out.close();
    System.out.println("Reading file");
    BufferedReader in2 = new BufferedReader(
      new InputStreamReader(new GZIPInputStream(
        new FileInputStream("test.gz"))));
    String s;
    while((s = in2.readLine()) != null)
      System.out.println(s);
  }
} /* (Da eseguire per visualizzare l'output) *///:~
```

L'utilizzo delle classi di compressione è molto semplice: è sufficiente inglobare il flusso di output in uno **GZIPOutputStream** o in uno **ZipOutputStream**, e il flusso di input in uno **GZIPInputStream** o in uno **ZipInputStream**; tutte le altre operazioni sono assimilabili alla normale lettura e scrittura dell'I/O. Questo è un esempio di miscelazione di flussi **char**-oriented con flussi **byte**-oriented; **in** si serve delle classi **Reader**, mentre il costruttore di **GZIPOutputStream** accetta soltanto un oggetto **OutputStream**. All'apertura del file, il flusso **GZIPInputStream** viene convertito in un **Reader**.

## *Creazione di archivi di file con Zip*

La libreria che supporta il formato Zip è più completa: vi consente di archiviare facilmente file multipli e include persino una classe separata per facilitare il processo di lettura di un archivio Zip. La libreria utilizza il formato standard Zip, in modo da operare in perfetta sintonia con tutti gli strumenti Zip disponibili su Internet. L'esempio seguente ha la stessa struttura di quello precedente, ma gestisce un numero arbitrario di argomenti da riga di comando; illustra inoltre l'utilizzo delle classi **Checksum** per il calcolo e la verifica delle somme di controllo dei file. Esistono due tipi di **Checksum**: **Adler32**, più rapida, e **CRC32**, più lenta ma più accurata.

```
//: io/ZipCompress.java
// Utilizzo della compressione Zip per comprimere un numero
// arbitrario di file elencati nella riga di comando.
// {Args: ZipCompress.java}
import java.util.zip.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ZipCompress {
  public static void main(String[] args)
  throws IOException {
    FileOutputStream f = new FileOutputStream("test.zip");
    CheckedOutputStream csum =
      new CheckedOutputStream(f, new Adler32());
     ZipOutputStream zos = new ZipOutputStream(csum);
     BufferedOutputStream out =
      new BufferedOutputStream(zos);
    zos.setComment("A test of Java Zipping");
    // Non esiste pero' nessun metodo getComment()
    // corrispondente.
    for(String arg : args) {
      print("Writing file " + arg);
      BufferedReader in =
        new BufferedReader(new FileReader(arg));
      zos.putNextEntry(new ZipEntry(arg));
      int c;
```

```
      while((c = in.read()) != -1)
        out.write(c);
      in.close();
      out.flush();
    }
    out.close();
    // La checksum e' valida soltanto dopo la chiusura del file!
    print("Checksum: " + csum.getChecksum().getValue());
    // Ora estrae i file:
    print("Reading file");
    FileInputStream fi = new FileInputStream("test.zip");
    CheckedInputStream csumi =
      new CheckedInputStream(fi, new Adler32());
    ZipInputStream in2 = new ZipInputStream(csumi);
    BufferedInputStream bis = new BufferedInputStream(in2);
    ZipEntry ze;
    while((ze = in2.getNextEntry()) != null) {
      print("Reading file " + ze);
      int x;
      while((x = bis.read()) != -1)
        System.out.write(x);
    }
    if(args.length == 1)
    print("Checksum: " + csumi.getChecksum().getValue());
    bis.close();
    // Tecnica alternativa per aprire e leggere i file Zip:
    ZipFile zf = new ZipFile("test.zip");
    Enumeration e = zf.entries();
    while(e.hasMoreElements()) {
      ZipEntry ze2 = (ZipEntry)e.nextElement();
      print("File: " + ze2);
      // ...ed estrae i dati come in precedenza
    }
    /* if(args.length == 1) */
  }
} /* (Da eseguire per visualizzare l'output) *///:~
```

Per ogni file da aggiungere all'archivio, dovete chiamare il metodo **putNextEntry()** passandogli un oggetto **ZipEntry**. Questo oggetto contiene un'interfaccia completa, che vi consente di ottenere e impostare tutti i dati disponibili per una specifica voce dell'archivio Zip: nomi, dimensioni compresse e non compresse, data, valore della checksum CRC, dati di campi supplementari, commento, metodo di compressione e indicazione se si tratta di una directory. Tuttavia, anche se il formato Zip standard consente di impostare una password, questa funzionalità non è supportata dalla libreria Zip di Java. Inoltre, nonostante **CheckedInputStream** e **CheckedOutputStream** siano compatibili con le checksum **Adler32** e **CRC32**, la classe **ZipEntry** supporta soltanto un'interfaccia per CRC: pur trattandosi di una limitazione del formato Zip sottostante, potrebbe comunque impedirvi di ricorrere al più veloce formato **Adler32**.

Per estrarre gli archivi, **ZipInputStream** offre il metodo **getNextEntry()**, il quale restituisce l'oggetto **ZipEntry** successivo, se presente. Come alternativa più concisa potete leggere l'archivio utilizzando un oggetto **ZipFile**, il cui metodo **entries()** restituisce un'**Enumeration** contenente le **ZipEntry**.

Per conoscere il totale di controllo dovete in qualche modo accedere all'oggetto **Checksum** collegato, in cui viene mantenuto un riferimento agli oggetti **CheckedInputStream** e **CheckedOutputStream**, ma potreste anche tenere semplicemente un riferimento all'oggetto **Checksum**.

Un metodo problematico dei flussi Zip è **setComment()**. Come mostrato in **ZipCompress.java**, potete impostare un commento in fase di scrittura dell'archivio, ma non avete modo di recuperare i commenti nel flusso **ZipInputStream**. I commenti sembrano essere supportati integralmente per ogni singola voce dell'archivio soltanto mediante **ZipEntry**.

Naturalmente non siete limitati agli archivi quando utilizzate le librerie **Zip** o **GZIP**: potete comprimere qualsiasi elemento, compresi i dati da trasmettere via rete.

## Archivi Java

Il formato Zip è utilizzato anche nel formato di archivio JAR (*Java ARchive*), che è un meccanismo di raggruppamento di file in un unico archivio compresso, del tutto analogo a Zip. Come tutto ciò che riguarda Java, anche gli archivi di JAR sono indipendenti dalla piattaforma, pertanto non dovete preoccuparvi di questo aspetto. Ricordate che, oltre ai file di classe, potete includere anche file di immagine e audio.

Gli archivi JAR sono utili soprattutto quando si lavora con Internet. Prima dell'introduzione degli archivi JAR, il browser doveva inviare ripetute richieste

al server web per trasferire tutti i componenti dell'applet, che peraltro erano file in formato naturale, non compresso. Combinando tutti i componenti di un determinato applet in un solo file JAR, invece, una sola richiesta al server è sufficiente e il trasferimento si rivela più veloce grazie alla compressione; inoltre, ogni file di un archivio JAR può avete una firma digitale di sicurezza.

Un archivio JAR si compone di un solo file contenente una raccolta di file compressi, con un "manifesto" che ne descrive le proprietà: potete creare voi stessi il vostro file-manifesto o, se preferite, lasciare che lo faccia il programma **jar**. Troverete maggiori informazioni sul manifesto JAR nella documentazione JDK.

Il programma di utilità **jar** fornito con il JDK di Sun comprime gli archivi di vostra scelta e può essere richiamato dalla riga di comando:

```
jar [options] destination [manifest] inputfile(s)
```

Le opzioni sono rappresentate semplicemente da caratteri, senza trattino o altri indicatori. Agli utenti che lavorano in ambienti UNIX/Linux non sarà sfuggita la somiglianza con le opzioni di **tar**:

| | |
|---|---|
| **c** | Crea un nuovo archivio o un archivio vuoto. |
| **t** | Elenca l'indice dei contenuti. |
| **x** | Estrae tutti i file. |
| **x *file*** | Estrae il file specificato. |
| **f** | Consente di specificare il nome del file JAR. Senza questa opzione, **jar** suppone che l'input proverrà da standard input oppure, in fase di creazione di un archivio, che l'output sarà destinato a standard output. |
| **m** | Indica che il primo argomento sarà il nome del file di manifesto creato dall'utente. |
| **v** | Produce una descrizione prolissa (*verbose*) dell'attività che **jar** sta eseguendo. |
| **0** (zero) | Registra soltanto i file, senza comprimerli. Questa opzione è utilizzata per creare un archivio JAR da includere nel CLASSPATH. |
| **M** | Non crea automaticamente un file di manifesto. |

Se tra gli elementi da includere nell'archivio JAR è presente una sottodirectory, verrà aggiunta automaticamente con tutte le sue sottodirectory e le relative informazioni di percorso.

Di seguito sono presentate alcune varianti di utilizzo di **jar**. Questo comando crea un file JAR chiamato **myJarFile.jar**, contenente tutti i file di classe presenti nella directory corrente, e un file di manifesto generato automaticamente:

```
jar cf myJarFile.jar *.class
```

Questo comando è analogo al precedente, ma aggiunge un file di manifesto creato dall'utente, chiamato **myManifestFile.mf**:

```
jar cmf myJarFile.jar myManifestFile.mf *.class
```

In questo caso, viene prodotto l'indice dei file contenuti in **myJarFile.jar**:

```
jar tf myJarFile.jar
```

Questo comando include l'opzione "verbose" per ottenere informazioni dettagliate sui file presenti in **myJarFile.jar**:

```
jar tvf myJarFile.jar
```

Presupponendo che **audio**, **classes** e **image** siano sottodirectory, questo comando le combina tutte nell'archivio **myApp.jar**. Viene passata anche l'opzione "verbose", per fornire informazioni supplementari durante l'esecuzione del programma **jar**:

```
jar cvf myApp.jar audio classes image
```

Se create un file JAR utilizzando l'opzione **0** (zero) potrete poi inserirlo nel CLASSPATH:

```
CLASSPATH="lib1.jar;lib2.jar;"
```

In questo modo, Java potrà cercare i file di classe in **lib1.jar** e **lib2.jar**. L'utility **jar** non è di impiego generale come il programma **Zip**. Per esempio, non potete aggiungere o aggiornare i file di un archivio JAR esistente, ma soltanto creare nuovi archivi JAR; inoltre, non è possibile spostare file presenti in un archivio JAR, cancellandoli durante lo spostamento. Un archivio JAR creato su una piattaforma può tuttavia essere letto senza problemi dall'utility **jar** di qualunque altra piattaforma, un inconveniente che invece compromette l'utilità di alcuni programmi di utilità **Zip**.

Come vedrete nel Volume 3, Capitolo 2, i file JAR sono utilizzati anche per "pacchettizzare" i JavaBeans.

## Serializzazione di oggetti

Quando create un oggetto, esso dura fino a quando sarà necessario, ma in nessun caso continua a esistere quando termina il programma. Se questo concetto inizialmente ha senso, vi sono anche situazioni in cui sarebbe utile che un oggetto potesse continuare a esistere e contenere informazioni anche quando il programma non è più in funzione: in questo modo, alla successiva esecuzione del programma l'oggetto esisterebbe ancora e conterrebbe le medesime informazioni che possedeva alla precedente esecuzione del programma. Naturalmente potete ottenere un risultato analogo registrando le informazioni su un file o in un database, ma nell'ottica di rendere tutto un oggetto sarebbe pratico dichiarare un oggetto come "persistente" e lasciare che sia il sistema a occuparsi dei dettagli.

La *serializzazione di oggetti* di Java vi consente di trasformare in una sequenza di byte qualsiasi oggetto implementi l'interfaccia **Serializable**; tali sequenze di byte potranno essere poi ripristinate, rigenerando l'oggetto originale. Questo è vero persino su una rete e sottintende che il meccanismo di serializzazione compensi automaticamente le differenze tra i vari sistemi operativi: in pratica, potete creare un oggetto su un sistema Windows, serializzarlo e inviarlo attraverso la rete a un sistema UNIX, dove l'oggetto sarà ricostruito correttamente. Non dovete preoccuparvi delle diverse rappresentazioni dei dati su sistemi differenti, né dell'ordinamento dei byte o di eventuali altri dettagli.

In sé, la serializzazione degli oggetti è interessante perché consente di implementare la cosiddetta *persistenza leggera* (*lightweight persistence*). "Persistenza" significa che il ciclo di vita di un oggetto non è determinato dal programma in esecuzione: l'oggetto continua a "vivere" *tra le sessioni* del programma. Scrivendo un oggetto serializzabile su disco e poi ripristinandolo quando il programma viene nuovamente eseguito, potete riprodurre l'effetto della persistenza. La ragione per cui è chiamata "leggera" è che non è possibile definire un oggetto utilizzando semplicemente, per esempio, un'ipotetica parola chiave "persistente" e lasciare che il sistema si prenda cura dei dettagli: forse tutto ciò sarà possibile in futuro, in ogni caso attualmente dovete serializzare e deserializzare in maniera esplicita gli oggetti nel vostro programma. Se vi occorrono funzionalità di persistenza più complete, potreste prendere in considerazione uno strumento quale Hibernate (*http://hibernate.sourceforge.net*). Per approfondire l'argomento consultate *Thinking in Enterprise Java*, scaricabile dal sito *www.mindview.net.*

La serializzazione degli oggetti è stata aggiunta al linguaggio per supportare due caratteristiche importanti. In primo luogo, la tecnologia (RMI) *Remote Method Invocation*, la quale consente agli oggetti che "vivono" su altri com-

puter di comportarsi come se esistessero sul vostro: quando inviate messaggi agli oggetti remoti, dovete ricorrere alla serializzazione degli oggetti per trasportare gli argomenti e i valori di ritorno. La tecnologia RMI è uno degli argomenti di *Thinking in Enterprise Java*.

La serializzazione degli oggetti è necessaria anche come supporto alla tecnologia JavaBeans, descritta nel Volume 3, Capitolo 2. Quando si utilizza un "bean" le informazioni del suo stato vengono configurate, generalmente in fase di progettazione. Tali informazioni devono essere memorizzate e poi recuperate all'avvio del programma; queste operazioni avvengono per mezzo della serializzazione.

La serializzazione di oggetti è ragionevolmente semplice purché l'oggetto implementi **Serializable**, un'interfaccia di tagging[6] che non dispone di alcun metodo. Quando in Java è stata integrata questa funzionalità, molte classi della libreria standard sono state modificate per renderle serializzabili, inclusi tutti i wrapper dei tipi primitivi, tutte le classi contenitore e molte altre; anche gli oggetti **Class** possono essere serializzati.

Per serializzare un oggetto, è necessario creare un oggetto di tipo **OutputStream** e incorporarlo in un flusso **ObjectOutputStream**; fatto questo, è sufficiente chiamare il metodo **writeObject()**: l'oggetto viene serializzato e inviato all'**OutputStream**. Ricordate che la serializzazione degli oggetti è di tipo **byte**-oriented, quindi si serve delle gerarchie **OutputStream** e **InputStream**. Il processo inverso richiede l'inglobamento di un **InputStream** in un **ObjectInputStream** e la chiamata al metodo **readObject()**: come di consueto, verrà restituito un riferimento a un **Object** sottoposto a upcast, pertanto sarà necessario eseguire il downcasting per ottenere l'oggetto originale.

Una funzionalità fondamentale della serializzazione di oggetti è che vengono salvati non solo l'immagine del vostro oggetto, ma anche *tutti* gli oggetti cui fa riferimento l'oggetto in questione; la serializzazione può anche continuare a seguire tutti i riferimenti in ciascuno di *questi* oggetti e così via. Talvolta si parla di questo insieme come della "ragnatela di oggetti" a cui un singolo oggetto può essere collegato, che include gli array di riferimenti agli oggetti e gli oggetti membro. Nel caso dobbiate gestire voi stessi uno schema di serializzazione, la manutenzione del codice necessario per seguire tutti questi collegamenti potrebbe diventare un'impresa titanica. La serializzazione di oggetti Java sembra tuttavia portarla a termine in modo impeccabile, senza dubbio grazie all'utilizzo di una procedura ottimizzata che attraversa l'intera "ragnatela" di oggetti. L'esempio seguente testa il meccanismo di serializza-

6. Interfaccia che serve esclusivamente ad attribuire a un oggetto un tipo verificabile usando la riflessione.

zione creando una struttura di oggetti collegati, ciascuno dei quali possiede un collegamento al segmento successivo nonché un array di riferimenti a oggetti di un'altra classe, **Data**:

```
//: io/Worm.java
// Dimostra la serializzazione di oggetti.
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Data implements Serializable {
  private int n;
  public Data(int n) { this.n = n; }
  public String toString() { return Integer.toString(n); }
}

public class Worm implements Serializable {
  private static Random rand = new Random(47);
  private Data[] d = {
    new Data(rand.nextInt(10)),
    new Data(rand.nextInt(10)),
    new Data(rand.nextInt(10))
  };
  private Worm next;
  private char c;
  // Il valore di i == numero di segmenti
  public Worm(int i, char x) {
    print("Worm constructor: " + i);
    c = x;
    if(--i > 0)
      next = new Worm(i, (char)(x + 1));
  }
  public Worm() {
    print("Default constructor");
  }
  public String toString() {
    StringBuilder result = new StringBuilder(":");
```

```
    result.append(c);
    result.append("(");
    for(Data dat : d)
      result.append(dat);
    result.append(")");
    if(next != null)
      result.append(next);
    return result.toString();
  }
  public static void main(String[] args)
  throws ClassNotFoundException, IOException {
    Worm w = new Worm(6, 'a');
    print("w = " + w);
    ObjectOutputStream out = new ObjectOutputStream(
      new FileOutputStream("worm.out"));
    out.writeObject("Worm storage\n");
    out.writeObject(w);
    out.close(); // Svuota anche l'output
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("worm.out"));
    String s = (String)in.readObject();
    Worm w2 = (Worm)in.readObject();
    print(s + "w2 = " + w2);
    ByteArrayOutputStream bout =
      new ByteArrayOutputStream();
    ObjectOutputStream out2 = new ObjectOutputStream(bout);
    out2.writeObject("Worm storage\n");
    out2.writeObject(w);
    out2.flush();
    ObjectInputStream in2 = new ObjectInputStream(
      new ByteArrayInputStream(bout.toByteArray()));
    s = (String)in2.readObject();
    Worm w3 = (Worm)in2.readObject();
    print(s + "w3 = " + w3);
  }
} /* Output:
```

```
Worm constructor: 6
Worm constructor: 5
Worm constructor: 4
Worm constructor: 3
Worm constructor: 2
Worm constructor: 1
w = :a(853):b(119):c(802):d(788):e(199):f(881)
Worm storage
w2 = :a(853):b(119):c(802):d(788):e(199):f(881)
Worm storage
w3 = :a(853):b(119):c(802):d(788):e(199):f(881)
*///:~
```

Per rendere le cose interessanti, l'array di oggetti **Data** all'interno di **Worm** viene inizializzato con numeri casuali; in questo modo non potrete sospettare che il compilatore conservi qualche genere di metainformazioni. Ogni segmento **Worm** è identificato tramite un valore **char**, che viene prodotto automaticamente nel processo di generazione ricorsiva dell'elenco collegato di oggetti **Worm**. Quando create un oggetto **Worm**, fornite al costruttore le sue dimensioni; per creare il riferimento **next**, il costruttore di **Worm** viene chiamato passandogli una lunghezza inferiore di un'unità rispetto al valore precedente ecc. Il riferimento **next** finale è lasciato al valore **null**, per indicare la fine del **Worm**.

L'obiettivo di questo esempio è creare qualcosa di sufficientemente complesso da non poter essere serializzato facilmente. La serializzazione in sé, tuttavia, è piuttosto semplice. Una volta che l'oggetto **ObjectOutputStream** è stato generato da qualche altro flusso, il metodo **writeObject()** serializza l'oggetto. Notate la chiamata a **writeObject()** anche per un oggetto **String**. Potete anche scrivere tutti i tipi di dato primitivi utilizzando gli stessi metodi di **DataOutputStream**, in quanto condividono la stessa interfaccia.

Notate che due sezioni del codice sembrano simili: la prima scrive e legge un file, mentre la seconda scrive e legge un **ByteArray**. Potete leggere e scrivere un oggetto utilizzando la serializzazione su qualsiasi **DataInputStream** o **DataOutputStream** (comprese le connessioni di rete), come potrete vedere in *Thinking in Enterprise Java*.

L'output mostra che l'oggetto deserializzato contiene effettivamente tutti i collegamenti che erano presenti nell'oggetto originale.

Notate che nel corso della deserializzazione di oggetti **Serializable** non viene chiamato alcun costruttore, neppure quello predefinito: l'intero oggetto è ripristinato recuperando i dati da **InputStream**.

> **Esercizio 27** (1) Create una classe **Serializable** contenente un riferimento a un oggetto di una seconda classe, anch'essa **Serializable**. Create un'istanza della vostra prima classe, serializzatela su disco, ripristinatela e verificate che l'intero processo si sia concluso correttamente.

## *Recupero di una classe*

Potreste domandarvi quali sono i requisiti necessari affinché un oggetto serializzato venga recuperato dallo stato serializzato. Per esempio, supponete di serializzare un oggetto e di trasmetterlo sotto forma di file o via rete a un altro computer. Un programma presente sul computer di destinazione sarà in grado di ricostruire l'oggetto basandosi unicamente sul contenuto del file?

La migliore risposta a questa domanda, come di consueto, è un esperimento. Il seguente file sorgente è presente nella sottodirectory degli esempi relativi a questo capitolo:

```
//: io/Alien.java
// Una classe serializzabile.
import java.io.*;
public class Alien implements Serializable {} ///:~
```

Il file che crea e serializza un oggetto **Alien** va posto nella stessa directory:

```
//: io/FreezeAlien.java
// Crea un file di output serializzato.
import java.io.*;

public class FreezeAlien {
  public static void main(String[] args) throws Exception {
    ObjectOutput out = new ObjectOutputStream(
      new FileOutputStream("X.file"));
    Alien quellek = new Alien();
    out.writeObject(quellek);
  }
} ///:~
```

Invece di intercettare e gestire le eccezioni, questo programma adotta il metodo (rapido per quanto "poco pulito") di passare le eccezioni all'esterno di **main()**, in modo che vengano visualizzate a console.

Quando il programma viene compilato ed eseguito produce un file chiamato **X.file** nella directory **io**. La classe seguente deve trovarsi in una sottodirectory di **io** chiamata **xfiles**:

```
//: io/xfiles/ThawAlien.java
// Cerca di ripristinare un file serializzato senza
// la classe dell'oggetto archiviato nel file.
// {RunByHand}
import java.io.*;

public class ThawAlien {
  public static void main(String[] args) throws Exception {
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream(new File("..", "X.file")));
    Object mystery = in.readObject();
    System.out.println(mystery.getClass());
  }
} /* Output:
class Alien
*///:~
```

Anche l'apertura del file e la lettura dell'oggetto **mistery** richiedono l'oggetto **Class** di **Alien**; la JVM non può trovare il file **Alien.class** a meno che non sia nel CLASSPATH (ipotesi non contemplata in questo esempio), pertanto solleva un'eccezione di tipo **ClassNotFoundException**. Ancora una volta, tutte le prove dell'esistenza di vita aliena spariscono prima di essere verificate! Come vedete, la JVM deve sempre essere in grado di trovare il file **.class** collegato.

## *Controllare la serializzazione*

Avete visto che il meccanismo di serializzazione predefinito è molto facile da utilizzare. Ma se aveste necessità specifiche? Potreste avere esigenze di sicurezza particolari, tali da non volere serializzare determinate porzioni del vostro oggetto, o semplicemente potrebbe non avere senso che una parte venga serializzata se la stessa parte deve essere ricreata *ex novo* quando l'oggetto viene recuperato.

Avete la possibilità di controllare il processo di serializzazione implementando l'interfaccia **Externalizable**, in luogo di **Serializable**. L'interfaccia **Externalizable** estende l'interfaccia **Serializable**, cui aggiunge due metodi, **writeExternal()** e **readExternal()**, che sono chiamati automaticamente per i vostri oggetti durante le operazioni di serializzazione e deserializzazione, per consentirvi di implementare operazioni specifiche.

L'esempio seguente mostra semplici implementazioni dei metodi di **Externalizable**; notate che le classi **Blip1** e **Blip2** sono quasi identiche, con una sottile differenza. Cercate di scoprirla analizzando il codice:

```
//: io/Blips.java
// Semplice utilizzo di Externalizable, con una trappola.
import java.io.*;
import static net.mindview.util.Print.*;

class Blip1 implements Externalizable {
  public Blip1() {
    print("Blip1 Constructor");
  }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    print("Blip1.writeExternal");
  }
  public void readExternal(ObjectInput in)
     throws IOException, ClassNotFoundException {
    print("Blip1.readExternal");
  }
}

class Blip2 implements Externalizable {
  Blip2() {
    print("Blip2 Constructor");
  }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    print("Blip2.writeExternal");
  }
  public void readExternal(ObjectInput in)
```

```
      throws IOException, ClassNotFoundException {
      print("Blip2.readExternal");
    }
  }

  public class Blips {
    public static void main(String[] args)
    throws IOException, ClassNotFoundException {
      print("Constructing objects:");
      Blip1 b1 = new Blip1();
      Blip2 b2 = new Blip2();
      ObjectOutputStream o = new ObjectOutputStream(
        new FileOutputStream("Blips.out"));
      print("Saving objects:");
      o.writeObject(b1);
      o.writeObject(b2);
      o.close();
      // Ripristino degli oggetti:
      ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("Blips.out"));
      print("Recovering b1:");
      b1 = (Blip1)in.readObject();
      // OOPS! Solleva un'eccezione:
  //! print("Recovering b2:");
  //! b2 = (Blip2)in.readObject();
    }
  } /* Output:
  Constructing objects:
  Blip1 Constructor
  Blip2 Constructor
  Saving objects:
  Blip1.writeExternal
  Blip2.writeExternal
  Recovering b1:
  Blip1 Constructor
  Blip1.readExternal
  *///:~
```

La ragione per cui l'oggetto **Blip2** non viene recuperato è che quando si tenta di farlo viene sollevata un'eccezione. Avete notato la differenza tra **Blip1** e **Blip2**? Il costruttore di **Blip1** è **public**, mentre quello di **Blip2** non lo è: questa differenza solleva l'eccezione in fase di ripristino. Per riprodurre il comportamento corretto, rendete **public** il costruttore di **Blip2** e rimuovete i commenti **//!**.

Quando **b1** viene recuperato, è anche chiamato il costruttore predefinito di **Blip1**. Questo comportamento è diverso dal recupero di un oggetto **Serializable**, in cui l'oggetto è interamente ricostruito dai bit memorizzati, senza chiamate al costruttore. Con un oggetto **Externalizable** si ha il comportamento predefinito di costruzione dell'oggetto, compresa l'inizializzazione dei campi, al quale *segue* la chiamata al metodo **readExternal()**. Dovete essere ben consci di questo, in particolare del fatto che tutte le operazioni predefinite di creazione dell'oggetto hanno sempre luogo, in modo da garantire il comportamento corretto dei vostri oggetti **Externalizable**.

L'esempio seguente mostra come archiviare e ripristinare interamente un oggetto **Externalizable**:

```
//: io/Blip3.java
// Ricostruzione di un oggetto Externalizable.
import java.io.*;
import static net.mindview.util.Print.*;

public class Blip3 implements Externalizable {
  private int i;
  private String s; // Nessuna inizializzazione
  public Blip3() {
    print("Blip3 Constructor");
    // s, i non inizializzati
  }
  public Blip3(String x, int a) {
    print("Blip3(String x, int a)");
    s = x;
    i = a;
    // s & i vengono inizializzati soltanto nel costruttore
    // non predefinito.
  }
  public String toString() { return s + i; }
  public void writeExternal(ObjectOutput out)
```

```
  throws IOException {
    print("Blip3.writeExternal");
    // Questo e' necessario:
    out.writeObject(s);
    out.writeInt(i);
  }
  public void readExternal(ObjectInput in)
  throws IOException, ClassNotFoundException {
    print("Blip3.readExternal");
    // Questo e' necessario:
    s = (String)in.readObject();
    i = in.readInt();
  }
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    print("Constructing objects:");
    Blip3 b3 = new Blip3("A Stringa ", 47);
    print(b3);
    ObjectOutputStream o = new ObjectOutputStream(
      new FileOutputStream("Blip3.out"));
    print("Saving object:");
    o.writeObject(b3);
    o.close();
    // Ripristino degli oggetti:
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("Blip3.out"));
    print("Recovering b3:");
    b3 = (Blip3)in.readObject();
    print(b3);
  }
} /* Output:
Constructing objects:
Blip3(String x, int a)
A String 47
Saving object:
Blip3.writeExternal
```

```
Recovering b3:
Blip3 Constructor
Blip3.readExternal
A String 47
*///:~
```

I campi **s** e **i** sono inizializzati soltanto nel secondo costruttore, non in quello predefinito; questo implica che se non inizializzate **s** e **i** in **readExternal()**, **s** sarà **null** e **i** zero: ricordate che la memoria riservata agli oggetti viene azzerata nella prima fase della creazione dell'oggetto. Se commentate le due righe di codice che seguono il commento **Questo e' necessario:** in **readExternal()** ed eseguite il programma, vedrete che al ripristino dell'oggetto **s** varrà **null** e **i** zero.

Se avete ereditato da un oggetto **Externalizable**, di norma chiamerete le versioni della classe di base dei metodi **writeExternal()** e **readExternal()** per eseguire le opportune operazioni di registrazione e ripristino dei componenti della classe di base.

Per ottenere il comportamento corretto, quindi, all'interno del metodo **writeExternal()** dovete assicurarvi di salvare i dati importanti dell'oggetto, perché non esiste un comportamento predefinito che salvi il contenuto degli oggetti membro di un oggetto **Externalizable**; nel metodo **readExternal()** dovete poi recuperare questi dati. Questa tecnica inizialmente può confondere, in quanto il comportamento predefinito di costruzione di un oggetto **Externalizable** potrebbe farlo sembrare un processo di registrazione e ripristino automatico, ma non è così.

**Esercizio 28** (2) Copiate il file **Blips.java** e modificatene il nome in **BlipCheck.java**; poi cambiate il nome della classe da **Blip2** a **BlipCheck** per renderla **public**, e rimuovete la dichiarazione di ambito *public* dalla classe **Blips**. Eliminate i commenti **//!** dal file ed eseguite il programma, poi commentate il costruttore predefinito di **BlipCheck**, eseguitelo e spiegate le ragioni del suo funzionamento. Tenete presente che dopo la compilazione dovrete eseguire il programma con il comando "**java Blips**", in quanto il metodo **main()** si troverà ancora nella classe **Blips**.

**Esercizio 29** (2) In **Blip3.java**, commentate le due righe poste dopo il commento **Questo e' necessario:** in **readExternal()** ed eseguite il programma. Illustrate il risultato e i motivi delle differenze riscontrate rispetto al programma originale.

## Parola chiave transient

Quando lavorate con la serializzazione, potreste trovarvi nella situazione di avere un sotto-oggetto particolare che non volete salvare e ripristinare automaticamente per mezzo del meccanismo di serializzazione Java: tale sotto-oggetto potrebbe contenere informazioni riservate che non è opportuno serializzare, come una password, per esempio. Anche se tali informazioni sono dichiarate **private** nell'oggetto, quando vengono sottoposte a serializzazione possono essere scoperte, leggendo un file o intercettando una trasmissione di rete.

Un metodo per impedire la serializzazione di porzioni riservate di un oggetto consiste, come si è visto, nell'implementare l'interfaccia **Externalizable**. In tal caso niente verrà serializzato automaticamente, e nel metodo **writeExternal()** potrete serializzare in modo esplicito soltanto le parti necessarie.

Se il vostro oggetto è **Serializable**, invece, la serializzazione avviene automaticamente. Per gestire questa situazione potete disattivare la serializzazione sui singoli campi servendovi della parola chiave **transient**, che esprime il concetto: "Non preoccuparti di salvare o ripristinare questo campo, me ne occupo io".

Per esempio, considerate un oggetto **Logon** che contiene informazioni su una particolare sessione di login; supponete che, una volta verificato il login, sia necessario registrare i dati ma non la password. Il modo più semplice per risolvere questo problema consiste nell'implementare **Serializable** e contrassegnare il campo **password** come **transient**. Ecco un esempio:

```
//: io/Logon.java
// Dimostra l'utilizzo della parola chiave "transient".
import java.util.concurrent.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Logon implements Serializable {
  private Date date = new Date();
  private String username;
  private transient String password;
  public Logon(String name, String pwd) {
    username = name;
    password = pwd;
  }
  public String toString() {
```

```
    return "logon info: \n username: " + username +
      "\n date: " + date + "\n password: " + password;
  }
  public static void main(String[] args) throws Exception {
    Logon a = new Logon("Hulk", "myLittlePony");
    print("login a = " + a);
    ObjectOutputStream o = new ObjectOutputStream(
      new FileOutputStream("Logon.out"));
    o.writeObject(a);
    o.close();
    TimeUnit.SECONDS.sleep(1); // Ritardo
    // Ripristino degli oggetti:
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("Logon.out"));
    print("Recovering object at " + new Date());
    a = (Logon)in.readObject();
    print("logon a = " + a);
  }
} /* Output: (Esempio)
logon a = logon info:
  username: Hulk
  date: Sat Nov 19 15:03:26 MST 2005
  password: myLittlePony
Recovering object at Sat Nov 19 15:03:28 MST 2005
logon a = logon info:
  username: Hulk
  date: Sat Nov 19 15:03:26 MST 2005
  password: null
*///:~
```

Potete vedere che i campi **date** e **username** sono normali (non **transient**), pertanto vengono serializzati automaticamente; il campo **password** è invece **transient**, quindi non viene memorizzato su disco; inoltre, il meccanismo di serializzazione non fa alcun tentativo per recuperare questo campo. Quando l'oggetto viene ripristinato, il campo **password** vale **null**. Tenete presente che, mentre il metodo **toString()** assembla un oggetto **String** utilizzando l'opera-

tore sovraccarico "**+**", un riferimento **null** viene automaticamente convertito nella stringa "null".

Il codice evidenzia anche il fatto che il campo **date** viene memorizzato su disco e poi ripristinato, e che quindi non è generato *ex novo*.

Dato che gli oggetti **Externalizable** in modalità predefinita non memorizzano campi, la parola chiave **transient** deve essere utilizzata soltanto con oggetti di **Serializable**.

### Un'alternativa all'interfaccia Externalizable

Se non ritenete opportuno implementare l'interfaccia **Externalizable**, esiste un metodo alternativo: potete implementare l'interfaccia **Serializable** e *aggiungere* i metodi **writeObject()** e **readObject()** che saranno chiamati in maniera automatica, rispettivamente, quando l'oggetto viene serializzato e deserializzato. È importante sottolineare che i metodi vengono "aggiunti", non "sovrascritti" o "implementati". Quindi, se fornite questi due metodi, essi verranno utilizzati in sostituzione della serializzazione predefinita.

I metodi devono avere esattamente queste segnature:

```
private void writeObject(ObjectOutputStream stream)
throws IOException;

private void readObject(ObjectInputStream stream)
throws IOException, ClassNotFoundException
```

Da un punto di vista progettuale, le cose diventano piuttosto strane. In primo luogo, potreste pensare che, dal momento che questi metodi non fanno parte di una classe di base o dell'interfaccia **Serializable**, devono essere definiti nella loro interfaccia. Notate, però, che i metodi sono definiti come **private**, il che significa che possono essere chiamati solo da altri membri della classe corrente. In realtà, tuttavia, *non* vengono chiamati da altri membri di questa classe: sono i metodi **writeObject()** e **readObject()** che fanno parte degli oggetti **ObjectInputStream** e **ObjectOutputStream** a chiamare i metodi **writeObject()** e **readObject()** del vostro oggetto. Potreste domandarvi come sia possibile che gli oggetti **ObjectInputStream** e **ObjectOutputStream** abbiano accesso ai metodi **private** della vostra classe. Si può soltanto supporre che questo meccanismo faccia parte della "magia" della serializzazione.[7]

---

7. Nel paragrafo "Interfacce e informazioni di tipo" del Capitolo 2 è stato descritto come accedere ai metodi **private** dall'esterno di una classe.

Tutto ciò che viene definito in un'interfaccia è automaticamente **public**, quindi se i metodi **writeObject()** e **readObject()** devono essere **private** non possono appartenere a un'interfaccia. Considerata la necessità di attenersi esattamente alle segnature suddette, l'effetto risultante è lo stesso dell'implementazione di un'interfaccia.

Potrebbe sembrare che, chiamando il metodo **ObjectOutputStream.writeObject()**, l'oggetto **Serializable** passato venga consultato, utilizzando senza dubbio la riflessione, per verificare se implementa il suo metodo **writeObject()**. In caso affermativo, viene saltato il normale processo di serializzazione e chiamato il metodo **writeObject()** personalizzato. Identiche considerazioni valgono per il metodo **readObject()**.

Ma c'è un'ulteriore svolta: all'interno del vostro metodo **writeObject()**, potete scegliere di eseguire l'operazione **writeObject()** predefinita chiamando **defaultWriteObject()**; analogamente, da **readObject()** potete chiamare **defaultReadObject()**. Di seguito è presentato un semplice esempio che mostra come controllare l'archiviazione e il ripristino di un oggetto **Serializable**:

```
//: io/SerialCtl.java
// Come controllare la serializzazione aggiungendo dei metodi
// writeObject() e readObject() personalizzati.
import java.io.*;

public class SerialCtl implements Serializable {
  private String a;
  private transient String b;
  public SerialCtl(String aa, String bb) {
    a = "Not transient: " + aa;
    b = "Transient: " + bb;
  }
  public String toString() { return a + "\n" + b; }
  private void writeObject(ObjectOutputStream stream)
  throws IOException {
    stream.defaultWriteObject();
    stream.writeObject(b);
  }
  private void readObject(ObjectInputStream stream)
  throws IOException, ClassNotFoundException {
    stream.defaultReadObject();
```

```
    b = (String)stream.readObject();
  }
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    SerialCtl sc = new SerialCtl("Test1", "Test2");
    System.out.println("Before:\n" + sc);
    ByteArrayOutputStream buf= new ByteArrayOutputStream();
    ObjectOutputStream o = new ObjectOutputStream(buf);
    o.writeObject(sc);
    // Ripristino degli oggetti:
    ObjectInputStream in = new ObjectInputStream(
      new ByteArrayInputStream(buf.toByteArray()));
    SerialCtl sc2 = (SerialCtl)in.readObject();
    System.out.println("After:\n" + sc2);
  }
} /* Output:
Before:
Not transient: Test1
Transient: Test2
After:
Not transient: Test1
Transient: Test2
*///:~
```

In questo codice sono presenti un campo **String** normale e un altro campo **String transient**, per dimostrare che il campo non **transient** viene salvato automaticamente con il metodo **defaultWriteObject()**, mentre il campo **transient** viene salvato e ripristinato esplicitamente. I campi vengono inizializzati nel costruttore e non durante la loro definizione, per evidenziare che non sono inizializzati da qualche meccanismo automatico nel corso della deserializzazione.

Se utilizzate il meccanismo predefinito per scrivere le porzioni non **transient** del vostro oggetto, dovrete chiamare **defaultWriteObject()** come prima operazione in **writeObject()** e **defaultReadObject()** come prima operazione in **readObject()**. Queste chiamate di metodo sono alquanto insolite: per esempio, potrebbe sembrare che stiate chiamando **defaultWriteObject()** per un **ObjectOutputStream** senza passare argomenti, ma che per qualche motivo il metodo conosca comunque il riferimento al vostro oggetto e sap-

pia come scrivere tutte le porzioni non **transient**. Un comportamento, a dir poco, "sinistro".

Il codice per la memorizzazione e il ripristino degli oggetti **transient** è decisamente più familiare. In **main()** viene creato un oggetto **SerialCtl**, poi serializzato a un flusso **ObjectOutputStream**. Notate come in questo caso ci si serva di un buffer, anziché di un file: per **ObjectOutputStream** il supporto è indifferente. La serializzazione avviene nella riga:

```
o.writeObject(sc);
```

Il metodo **writeObject()** deve esaminare **sc** per controllare che disponga di un suo metodo **writeObject()**: il controllo non si verifica nell'interfaccia (che non esiste), né avviene sulla base del tipo di classe, ma "ricercando" effettivamente il metodo tramite la riflessione: se il metodo esiste, verrà utilizzato. Un approccio simile è adottato per **readObject()**. È possibile che questo sia stato l'unico modo pratico con il quale i tecnici di Sun abbiano potuto risolvere il problema, ma in ogni caso è certamente inconsueto.

### Gestione delle versioni

È possibile che vogliate cambiare la versione di una classe serializzabile, per esempio per memorizzare oggetti della classe originale in un database. Questa funzionalità è supportata, anche se probabilmente ve ne servirete soltanto in casi speciali; richiede inoltre uno sforzo supplementare per comprendere nozioni avanzate che non si è ritenuto opportuno fornire in questa sede: tenete presente che la documentazione JDK sull'argomento (scaricabile da *http://java.sun.com*) è completa.

Consultando la documentazione JDK, noterete anche numerosi commenti che iniziano con:

> ***Attenzione:*** *gli oggetti serializzati di questa classe non saranno compatibili con le future versioni di Swing. Il supporto alla serializzazione corrente è appropriato per la memorizzazione a breve termine o per le chiamate RMI (Remote Method Invocation) tra le applicazioni…*

Tali segnalazioni sono state inserite perché il meccanismo di versioning è troppo semplice per funzionare in modo attendibile in qualsiasi situazione, soprattutto con i JavaBeans. I tecnici di Sun stanno lavorando per porre rimedio al problema, ed è a questo che si riferiscono tali avvertimenti.

## Utilizzo della persistenza

È affascinante l'idea di utilizzare la tecnologia di serializzazione per memorizzare alcuni stati del programma, in modo da ristabilire in seguito il programma alla condizione desiderata. Tuttavia prima di usufruire di questa funzionalità, è necessario rispondere ad alcune domande. Che cosa accade serializzando due oggetti che puntano entrambi a un terzo oggetto? Quando ripristinerete i due oggetti dal loro stato serializzato otterrete soltanto un'occorrenza del terzo oggetto? Che cosa accade se serializzate i due oggetti in file separati e li deserializzate in diversi punti del codice?

Ecco un esempio che illustra il problema:

```
//: io/MyWorld.java
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

class House implements Serializable {}

class Animal implements Serializable {
  private String name;
  private House preferredHouse;
  Animal(String nm, House h) {
    name = nm;
    preferredHouse = h;
  }
  public String toString() {
    return name + "[" + super.toString() +
      "], " + preferredHouse + "\n";
  }
}

public class MyWorld {
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    House house = new House();
    List<Animal> animals = new ArrayList<Animal>();
    animals.add(new Animal("Bosco the dog", house));
```

```
    animals.add(new Animal("Ralph the hamster", house));
    animals.add(new Animal("Molly the cat", house));
    print("animals: " + animals);
    ByteArrayOutputStream buf1 =
      new ByteArrayOutputStream();
    ObjectOutputStream o1 = new ObjectOutputStream(buf1);
    o1.writeObject(animals);
    o1.writeObject(animals); // Scrive un secondo insieme
    // Scrive su un altro flusso:
    ByteArrayOutputStream buf2 =
      new ByteArrayOutputStream();
    ObjectOutputStream o2 = new ObjectOutputStream(buf2);
    o2.writeObject(animals);
    // Ora richiama gli oggetti:
    ObjectInputStream in1 = new ObjectInputStream(
      new ByteArrayInputStream(buf1.toByteArray()));
    ObjectInputStream in2 = new ObjectInputStream(
      new ByteArrayInputStream(buf2.toByteArray()));
    List
      animals1 = (List)in1.readObject(),
      animals2 = (List)in1.readObject(),
      animals3 = (List)in2.readObject();
    print("animals1: " + animals1);
    print("animals2: " + animals2);
    print("animals3: " + animals3);
  }
} /* Output:
animals: [Bosco the dog[Animal@addbf1], House@42e816,
Ralph the hamster[Animal@9304b1], House@42e816,
Molly the cat[Animal@190d11], House@42e816
]
animals1: [Bosco the dog[Animal@de6f34], House@156ee8e,
Ralph the hamster[Animal@47b480], House@156ee8e,
Molly the cat[Animal@19b49e6], House@156ee8e
]
animals2: [Bosco the dog[Animal@de6f34], House@156ee8e,
```

```
Ralph the hamster[Animal@47b480], House@156ee8e,
Molly the cat[Animal@19b49e6], House@156ee8e
]
animals3: [Bosco the dog[Animal@10d448], House@e0e1c6,
Ralph the hamster[Animal@6ca1c], House@e0e1c6,
Molly the cat[Animal@1bf216a], House@e0e1c6
]
*///:~
```

Un aspetto interessante di questo codice è la possibilità di utilizzare la serializzazione dell'oggetto verso e da un array di **byte** come mezzo per realizzare una "copia profonda" di qualsiasi oggetto **Serializable**: con l'espressione "copia profonda" si intende la duplicazione dell'intera "ragnatela" di oggetti, non soltanto dell'oggetto di base e dei suoi riferimenti. La copia degli oggetti è uno degli argomenti approfonditi nei supplementi online di questo volume.

Gli oggetti **Animal** contengono campi di tipo **House**. In **main()** viene creata una **List** di questi **Animal**, poi serializzata per due volte in un flusso e ancora una volta in un flusso separato. Quando questi flussi vengono deserializzati e visualizzati, si ottiene l'output riferito a una sola deserializzazione; tenete presente che gli oggetti saranno ogni volta in posizioni di memoria diverse.

Naturalmente, potreste aspettarvi che gli oggetti deserializzati abbiano indirizzi diversi da quelli originali. Come potete notare, nell'output di **animals1** (per **animals1**) e **animals2** (per **animals2**) vengono visualizzati gli stessi indirizzi, compresi i riferimenti all'oggetto **House** condiviso da entrambi gli oggetti. D'altra parte, quando recupera **animals3** il programma non ha modo di sapere che gli oggetti in questo altro flusso sono alias degli oggetti nel primo flusso, pertanto restituisce una struttura di oggetti completamente diversa.

Finché si tratta di serializzare ogni cosa in un solo flusso, otterrete la stessa struttura di oggetti che avete scritto, senza alcuna duplicazione accidentale di oggetti. Ovviamente, nulla vieta di cambiare lo stato degli oggetti nel tempo che intercorre tra la scrittura del primo e dell'ultimo oggetto, ma questa è responsabilità del programmatore; gli oggetti vengono scritti nello stato in cui si trovano, qualunque esso sia e con tutte le connessioni ad altri oggetti, nel momento in cui vengono serializzati.

La soluzione più sicura quando volete salvare la condizione di un sistema è serializzarla come operazione "atomica". Se serializzate alcuni oggetti, eseguite alcune operazioni, poi serializzate altri oggetti e così via, non memorizzerete la condizione del sistema in modo sicuro. È preferibile invece inserire tutti gli oggetti che fanno parte della condizione di sistema in un solo

contenitore, poi registrare questo contenitore in una sola operazione. A quel punto, potrete ripristinarlo con un'unica chiamata di metodo.

L'esempio seguente si riferisce a un ipotetico sistema CAD che dimostra questo approccio, evidenziando anche il problema dei campi **static**. Esaminando la documentazione JDK vedrete che **Class** è **Serializable**, pertanto non dovrebbero esserci problemi di memorizzazione dei campi **static** semplicemente serializzando l'oggetto **Class**: questa, quantomeno, sembra essere una tecnica ragionevole.

```
//: io/StoreCADState.java
// Come salvare lo stato di un ipotetico sistema CAD.
import java.io.*;
import java.util.*;

abstract class Shape implements Serializable {
  public static final int RED = 1, BLUE = 2, GREEN = 3;
  private int xPos, yPos, dimension;
  private static Random rand = new Random(47);
  private static int counter = 0;
  public abstract void setColor(int newColor);
  public abstract int getColor();
  public Shape(int xVal, int yVal, int dim) {
    xPos = xVal;
    yPos = yVal;
    dimension = dim;
  }
  public String toString() {
    return getClass() +
      "color[" + getColor() + "] xPos[" + xPos +
      "] yPos[" + yPos + "] dim[" + dimension + "]\n";
  }
  public static Shape randomFactory() {
    int xVal = rand.nextInt(100);
    int yVal = rand.nextInt(100);
    int dim = rand.nextInt(100);
    switch(counter++ % 3) {
      default:
```

```
      case 0: return new Circle(xVal, yVal, dim);
      case 1: return new Square(xVal, yVal, dim);
      case 2: return new Line(xVal, yVal, dim);
    }
  }
}

class Circle extends Shape {
  private static int color = RED;
  public Circle(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
  }
  public void setColor(int newColor) { color = newColor; }
  public int getColor() { return color; }
}

class Square extends Shape {
  private static int color;
  public Square(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
    color = RED;
  }
  public void setColor(int newColor) { color = newColor; }
  public int getColor() { return color; }
}

class Line extends Shape {
  private static int color = RED;
  public static void
  serializeStaticState(ObjectOutputStream os)
  throws IOException { os.writeInt(color); }
  public static void
  deserializeStaticState(ObjectInputStream os)
  throws IOException { color = os.readInt(); }
  public Line(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
```

```
  }
  public void setColor(int newColor) { color = newColor; }
  public int getColor() { return color; }
}

public class StoreCADState {
  public static void main(String[] args) throws Exception {
    List<Class<? extends Shape>> shapeTypes =
      new ArrayList<Class<? extends Shape>>();
    // Aggiunge i riferimenti agli oggetti di classe:
    shapeTypes.add(Circle.class);
    shapeTypes.add(Square.class);
    shapeTypes.add(Line.class);
    List<Shape> shapes = new ArrayList<Shape>();
    // Disegna alcune forme:
    for(int i = 0; i < 10; i++)
      shapes.add(Shape.randomFactory());
    // Imposta tutti i colori statici a verde (GREEN):
    for(int i = 0; i < 10; i++)
      ((Shape)shapes.get(i)).setColor(Shape.GREEN);
    // Salva il vettore di stato:
    ObjectOutputStream out = new ObjectOutputStream(
      new FileOutputStream("CADState.out"));
    out.writeObject(shapeTypes);
    Line.serializeStaticState(out);
    out.writeObject(shapes);
    // Visualizza le forme:
    System.out.println(shapes);
  }
} /* Output:
[class Circlecolor[3] xPos[58] yPos[55] dim[93]
, class Squarecolor[3] xPos[61] yPos[61] dim[29]
, class Linecolor[3] xPos[68] yPos[0] dim[22]
, class Circlecolor[3] xPos[7] yPos[88] dim[28]
, class Squarecolor[3] xPos[51] yPos[89] dim[9]
, class Linecolor[3] xPos[78] yPos[98] dim[61]
```

```
, class Circlecolor[3] xPos[20] yPos[58] dim[16]
, class Squarecolor[3] xPos[40] yPos[11] dim[22]
, class Linecolor[3] xPos[4] yPos[83] dim[6]
, class Circlecolor[3] xPos[75] yPos[10] dim[42]
]
*///:~
```

La classe **Shape** implementa **(implements) Serializable**, pertanto qualunque oggetto venga ereditato da **Shape** è anche automaticamente **Serializable**. Ogni **Shape** contiene i dati, e ogni classe **Shape** derivata contiene un campo **static** che determina il colore di tutti quei tipi di **Shape**. Ricordate che la creazione di un campo **static** nella classe di base darebbe luogo a un solo campo, dal momento che i campi **static** non vengono duplicati nelle classi derivate. I metodi nella classe di base possono essere sovrascritti per impostare il colore dei vari tipi: i metodi **static** non sono soggetti al polimorfismo (*binding dinamico*), pertanto sono metodi normali. Il metodo **randomFactory()** crea una forma **Shape** diversa a seguito di ogni chiamata, utilizzando valori casuali per i dati di **Shape**.

**Circle** e **Square** sono estensioni dirette di **Shape**; l'unica differenza è che **Circle** inizializza **color** al momento della definizione, mentre **Square** lo inizializza nel costruttore. L'oggetto **Line** verrà esaminato in seguito.

In **main()**, uno dei due **ArrayList** è utilizzato per contenere gli oggetti **Class** e l'altro per contenere le forme.

Il recupero degli oggetti è piuttosto semplice:

```
//: io/RecoverCADState.java
// Come ripristinare lo stato di un ipotetico sistema CAD.
// {RunFirst: StoreCADState}
import java.io.*;
import java.util.*;

public class RecoverCADState {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) throws Exception {
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("CADState.out"));
    // Legge gli oggetti serializzati nello stesso ordine
    // in cui sono stati scritti:
```

```
    List<Class<? extends Shape>> shapeTypes =
      (List<Class<? extends Shape>>)in.readObject();
    Line.deserializeStaticState(in);
    List<Shape> shapes = (List<Shape>)in.readObject();
    System.out.println(shapes);
  }
} /* Output:
[class Circlecolor[1] xPos[58] yPos[55] dim[93]
, class Squarecolor[0] xPos[61] yPos[61] dim[29]
, class Linecolor[3] xPos[68] yPos[0] dim[22]
, class Circlecolor[1] xPos[7] yPos[88] dim[28]
, class Squarecolor[0] xPos[51] yPos[89] dim[9]
, class Linecolor[3] xPos[78] yPos[98] dim[61]
, class Circlecolor[1] xPos[20] yPos[58] dim[16]
, class Squarecolor[0] xPos[40] yPos[11] dim22]
, class Linecolor[3] xPos[4] yPos[83] dim[6]
, class Circlecolor[1] xPos[75] yPos[10] dim[42]
]
*///:~
```

Come potete notare, i valori di **xPos**, **yPos** e **dim** sono stati tutti memorizzati e recuperati con successo, ma vi è un problema nel recupero delle informazioni **static**: i valori salvati valevano tutti "3", ma una volta ripristinati assumono valori diversi.

Le forme **Circle** hanno valore 1 (vale a dire **RED**, la sua definizione) e gli **Square** valore 0, perché vengono inizializzati nel costruttore. È come se i campi **static** non fossero stati serializzati! In effetti, sebbene sia **Serializable** la classe **Class** non si comporta nel modo previsto, pertanto se volete serializzare i campi **static** dovrete farlo voi stessi.

È a questo che servono i metodi **static serializeStaticState()** e **deserializeStaticState()** in **Line**, che vengono chiamati esplicitamente come componenti dei processi di memorizzazione e di recupero. Tenete presente che l'ordine di scrittura nel file di serializzazione e di lettura deve essere mantenuto scrupolosamente. Di conseguenza, per eseguire in modo corretto questi programmi dovete seguire la procedura descritta di seguito.

1. Aggiungere i metodi **serializeStaticState()** e **deserializeStaticState()** alle diverse forme.
2. Eliminare l'**ArrayList shapeTypes** e tutto il codice che vi fa riferimento.

3. Aggiungere le chiamate ai nuovi metodi **static** di serializzazione e deserializzazione nelle forme.

Considerato che la serializzazione salva anche i dati **private**, un altro tema cui dovreste pensare è la sicurezza. Se avete campi caratterizzati da esigenze di questo tipo, dovreste contrassegnarli come **transient**; a quel punto, però, dovrete progettare un meccanismo sicuro per la memorizzazione delle informazioni in modo che, al momento del ripristino, tali campi vengano ripristinati come variabili **private**.

**Esercizio 30** (1) Modificate il programma **CADState.java** come descritto nel testo.

## XML

Un limite importante alla serializzazione degli oggetti è che si tratta di una soluzione valida unicamente per Java: soltanto i programmi Java sono in grado di deserializzare tali oggetti. Una soluzione più aperta consiste nel convertire i dati in XML, un formato utilizzabile da un gran numero di piattaforme e di linguaggi.

A causa della sua popolarità, XML è supportato da una pletora di opzioni e utility specifiche per la programmazione in questo linguaggio, incluse le librerie **javax.xml.*** distribuite con il JDK di Java.

L'autore ha scelto di utilizzare la libreria open source XOM di Elliotte Rusty Harold, disponibile per il download all'indirizzo *www.xom.nu*, completa della relativa documentazione: l'autore ritiene infatti che questo software sia lo strumento più semplice e diretto per generare e modificare codice XML in Java, considerato anche il modo in cui enfatizza la precisione di XML.

Come esempio, supponete di avere alcuni oggetti **Person** contenenti nomi e cognomi che volete serializzare in formato XML.

La classe **Person** mostrata di seguito ha un metodo **getXML()** che utilizza XOM per produrre i dati **Person** convertiti in un oggetto XML **Element**, e un costruttore che accetta un oggetto **Element** e ne estrae i dati di **Person** opportuni (gli esempi di XML sono presenti nelle rispettive sottodirectory):

```
//: xml/Person.java
// Utilizzo della libreria XOM per scrivere e leggere dati XML
// {Richiede: nu.xom.Node; richiede l'installazione della
```

```
// libreria XOM (http://www.xom.nu) }
import nu.xom.*;
import java.io.*;
import java.util.*;

public class Person {
  private String first, last;
  public Person(String first, String last) {
    this.first = first;
    this.last = last;
  }
  // Produce un Element XML Element dall'oggetto Person
  // corrente:
  public Element getXML() {
    Element person = new Element("person");
    Element firstName = new Element("first");
    firstName.appendChild(first);
    Element lastName = new Element("last");
    lastName.appendChild(last);
    person.appendChild(firstName);
    person.appendChild(lastName);
    return person;
  }
  // Construttore che ripristina una Person da un Element XML:
  public Person(Element person) {
    first= person.getFirstChildElement("first").getValue();
    last = person.getFirstChildElement("last").getValue();
  }
  public String toString() { return first + " " + last; }
  // Formattazione per i lettori umani:
  public static void
  format(OutputStream os, Document doc) throws Exception {
    Serializer serializer= new Serializer(os,"ISO-8859-1");
    serializer.setIndent(4);
    serializer.setMaxLength(60);
    serializer.write(doc);
```

```
      serializer.flush();
    }
    public static void main(String[] args) throws Exception {
      List<Person> people = Arrays.asList(
        new Person("Dr. Bunsen", "Honeydew"),
        new Person("Gonzo", "The Great"),
        new Person("Phillip J.", "Fry"));
      System.out.println(people);
      Element root = new Element("people");
      for(Person p : people)
        root.appendChild(p.getXML());
      Document doc = new Document(root);
      format(System.out, doc);
      format(new BufferedOutputStream(new FileOutputStream(
        "People.xml")), doc);
    }
  } /* Output:
  [Dr. Bunsen Honeydew, Gonzo The Great, Phillip J. Fry]
  <?xml version="1.0" encoding="ISO-8859-1"?>
  <people>
      <person>
          <first>Dr. Bunsen</first>
          <last>Honeydew</last>
      </person>
      <person>
          <first>Gonzo</first>
          <last>The Great</last>
      </person>
      <person>
          <first>Phillip J.</first>
          <last>Fry</last>
      </person>
  </people>
  *///:~
```

I metodi di XOM sono ragionevolmente chiari e ben descritti nella documentazione della libreria.

XOM contiene anche una classe **Serializer**, che potete vedere in utilizzo nel metodo **format()**, per trasformare il codice XML in una forma più leggibile. Se vi limitate a chiamare **toXML()**, otterrete un funzionamento uniforme, e questo è il vantaggio della classe **Serializer** in termini di praticità.

La deserializzazione degli oggetti **Person** da un file XML è altrettanto semplice:

```
//: xml/People.java
// {Richiede: nu.xom.Node; richiede l'installazione della
// libreria XOM (http://www.xom.nu) }
// {RunFirst: Person}
import nu.xom.*;
import java.util.*;

public class People extends ArrayList<Person> {
  public People(String fileName) throws Exception  {
    Document doc = new Builder().build(fileName);
    Elements elements =
      doc.getRootElement().getChildElements();
    for(int i = 0; i < elements.size(); i++)
      add(new Person(elements.get(i)));
  }
  public static void main(String[] args) throws Exception {
    People p = new People("People.xml");
    System.out.println(p);
  }
} /* Output:
[Dr. Bunsen Honeydew, Gonzo The Great, Phillip J. Fry]
*///:~
```

Il costruttore di **People** apre e legge un file ricorrendo al metodo **Builder.build()** di XOM, e il metodo **getChildElements()** produce un elenco di **Element**: non si tratta di una **List** standard di Java, ma di un oggetto che possiede soltanto i metodi **size()** e **get()**, dal momento che l'autore della libreria non ha voluto forzare i programmatori a utilizzare Java SE5, pur volendo implementare un contenitore di tipo *type-safe*. Ogni **Element** in questo elenco rappresenta un oggetto **Person**, pertanto viene passato al secondo costruttore di **Person**. Notate che questa operazione implica la conoscenza preliminare della strut-

tura esatta dell'archivio XML, come spesso avviene quando si ha a che fare con questo genere di problemi. Se la struttura non corrisponde alle vostre aspettative, XOM solleverà un'eccezione. Avete anche la possibilità di scrivere codice più complesso che analizzi il documento XML anziché darne per scontate le caratteristiche, per le situazioni in cui non abbiate sufficienti informazioni sulla struttura del documento XML in ingresso.

Per fare in modo che questi esempi compilino, dovrete includere i file JAR della distribuzione XOM nel vostro CLASSPATH.

Questa è soltanto una rapida introduzione alla programmazione XML in Java e alla libreria XOM; sul sito *www.xom.nu* troverete maggiori informazioni.

> **Esercizio 31**: (2) Aggiungete informazioni di indirizzo adatte a **Person.java** e a **People.java**.
>
> **Esercizio 32** (4) Servendovi di **Map<String, Integer>** e dell'utility **net.mindview.util.TextFile**, scrivete un programma che conti le occorrenze delle parole presenti in un file: usate "**\\W+**" come secondo argomento per il costruttore di **TextFile**. Memorizzate i risultati sotto forma di documento XML.

## API Preferences

Le funzionalità offerte dalle API *Preferences* sono molto più vicine al concetto di persistenza che alla serializzazione degli oggetti, in quanto registrano e recuperano automaticamente le vostre informazioni. Tuttavia, il loro utilizzo si riduce a dataset di dimensioni limitate: queste API possono contenere soltanto i tipi primitivi e le **String**, e la lunghezza di ogni **String** memorizzata non può eccedere gli 8 KB, un valore non piccolissimo ma generalmente insufficiente per sviluppare qualcosa di serio. Come suggerisce il nome, le API Preferences consentono la memorizzazione e il ripristino delle preferenze dell'utente, nonché delle impostazioni di configurazione dei programmi.

Con una tecnica analoga a quella delle **Map**, le preferenze sono costituite da coppie di chiave-valore registrate in una gerarchia di nodi. Benché questo tipo di gerarchia possa essere utilizzato per creare strutture complesse, è in genere adottato per generare un solo nodo con lo stesso nome della classe che rappresenta, al cui interno è possibile memorizzare le informazioni. Considerate questo semplice esempio:

```
//: io/PreferencesDemo.java
import java.util.prefs.*;
import static net.mindview.util.Print.*;
```

```
public class PreferencesDemo {
  public static void main(String[] args) throws Exception {
    Preferences prefs = Preferences
      .userNodeForPackage(PreferencesDemo.class);
    prefs.put("Location", "Oz");
    prefs.put("Footwear", "Ruby Slippers");
    prefs.putInt("Companions", 4);
    prefs.putBoolean("Are there witches?", true);
    int usageCount = prefs.getInt("UsageCount", 0);
    usageCount++;
    prefs.putInt("UsageCount", usageCount);
    for(String key : prefs.keys())
      print(key + ": "+ prefs.get(key, null));
    // Occorre sempre fornire un valore predefinito:
    print("How many companions does Dorothy have? " +
      prefs.getInt("Compagni", 0));
  }
} /* Output: (Esempio)
Location: Oz
Footwear: Ruby Slippers
Companions: 4
Are there witches?: true
UsageCount: 53
How many companions does Dorothy have? 4
*///:~
```

In questo caso viene impiegato il metodo **userNodeForPackage()**, ma potreste anche optare per **systemNodeForPackage()**; la scelta è in un certo senso arbitraria, ma il concetto è che "user" si riferisce alle diverse preferenze dell'utente e "system" a quelle generali di configurazione dell'installazione. Dal momento che il metodo **main()** è **static**, per identificare il nodo viene utilizzata la classe **PreferencesDemo.class**, tuttavia nell'ambito di un metodo non **static** vi servirete di norma di **getClass()**. Non è indispensabile utilizzare la classe corrente come identificativo del nodo, benché questa sia la prassi abituale.

Una volta creato, il nodo è disponibile per il caricamento o la lettura dei dati. Questo esempio carica dapprima il nodo con vari tipi di elementi, poi ottiene le chiavi con **keys()**; queste chiavi vengono restituite sotto forma di **String[]**, che potrebbe essere una novità per chi è abituato al metodo **keys()** della libre-

ria di Collection. Notate il secondo argomento di **get()**: è il valore predefinito che viene prodotto in assenza di valore per la chiave corrente. Iterando un insieme di chiavi, sapete sempre che c'è una chiave, pertanto l'utilizzo di **null** come valore predefinito in questo caso specifico è una pratica considerata sicura; di solito, tuttavia, richiederete una chiave nominativa, come in:

```
prefs.getInt("Companions", 0));
```

Solitamente, vorrete fornire un valore predefinito adeguato. In effetti, una tipica forma idiomatica è quella dell'esempio seguente:

```
int usageCount = prefs.getInt("UsageCount", 0);
    usageCount++;
    prefs.putInt("UsageCount", usageCount);
```

Così facendo, la prima volta che eseguirete il programma **UsageCount** sarà zero, mentre nelle successive invocazioni sarà diverso da zero. Ogni volta che eseguite **PreferencesDemo.java**, noterete che **UsageCount** effettivamente incrementa il suo valore. Ma dove viene memorizzato questo valore? Non esiste alcun file locale che venga creato dopo la prima esecuzione del programma.

Le API Preferences utilizzano le risorse di sistema adatte per svolgere le proprie operazioni, e tali risorse variano in funzione del sistema operativo; in Windows, per esempio, viene utilizzato il registro di sistema, che già dispone di una gerarchia di nodi con coppie di chiave e valore. Comunque sia, le informazioni sono memorizzate "magicamente" per voi, e quindi non dovete preoccuparvi di come viene implementato questo meccanismo nei vari sistemi.

L'argomento delle API Preferences non si esaurisce qui: nella documentazione JDK, ben scritta, troverete tutti i dettagli che vi occorrono.

> **Esercizio 33** (2) Scrivete un programma che visualizzi il valore corrente di una directory chiamata "directory di base" e chieda di fornire un nuovo valore. Utilizzate le API Preferences per memorizzare il valore.

## Riepilogo

La libreria di flussi I/O Java risponde alle esigenze di base, consentendo di effettuare operazioni di lettura e scrittura su console, su un file, un blocco di memoria e persino via Internet. Grazie all'ereditarietà, vi consente di creare nuovi tipi di oggetti di input e output; potete persino estendere entro certi

limiti i tipi di oggetti accettati da un flusso, ridefinendo il metodo **toString()** che viene chiamato automaticamente quando passate un oggetto a un metodo che si aspetta una **String**, con la limitata "conversione di tipo automatico" di Java.

La documentazione e il progetto della libreria di I/O, tuttavia, lasciano aperte alcune questioni: per esempio, sarebbe stato pratico poter sollevare un'eccezione in caso di sovrascrittura di un file al momento dell'apertura per l'output, come avviene in alcuni linguaggi che vi consentono di utilizzare un file soltanto nel caso in cui non esista già. In Java apparentemente è necessario servirsi di un oggetto **File** per determinare se un file esiste, perché aprendolo come **FileOutputStream** o **FileWriter** verrà sempre sovrascritto.

È vero che la libreria di flussi I/O svolge la maggior parte del lavoro ed è portatile, ma è altrettanto vero che se non si comprende a fondo il design pattern Decorator, il progetto non risulta affatto intuitivo, ed è quindi causa di oneri aggiuntivi legati al suo apprendimento. Inoltre, la libreria non è completa: se lo fosse stata, l'autore non avrebbe dovuto realizzare programmi di utilità come **TextFile**; certo, la nuova classe **PrintWriter** di Java SE5 è un passo nella giusta direzione, ma per il momento rappresenta soltanto una soluzione parziale. Java SE5 ha comunque apportato notevoli miglioramenti, supportando quelle funzionalità di formattazione dell'output che praticamente ogni linguaggio ha sempre supportato.

*Quando* comprenderete il modello Decorator e inizierete a servirvi della libreria in situazioni che richiedono una relativa flessibilità, comincerete a trarre vantaggio da questo progetto, e a quel punto l'onere relativo rappresentato da qualche riga di codice in più potrebbe non costituire un problema.

> *La soluzione degli esercizi è disponibile nel documento The Thinking in Java Annotated Solution Guide, in vendita all'indirizzo www.mindview.net.*

# Capitolo 7

# Tipi enumerativi

La parola chiave **enum** consente di creare un nuovo tipo con un insieme limitato di valori e di considerare quei valori come normali componenti del programma, una tecnica particolarmente utile.[1]

I tipi enumerativi sono già stati introdotti brevemente al termine nel Volume 1, Capitolo 5, ma ora che avete compreso alcuni degli argomenti più complessi di Java potete analizzare i dettagli di questa funzionalità di Java SE5.

Non soltanto vedrete che le **enum** offrono alcune potenzialità molto interessanti, ma in questo capitolo avrete anche modo di approfondire ulteriormente altre caratteristiche di Java che ormai conoscete, quali i generici e la riflessione, nonché di esaminare alcuni nuovi esempi di design pattern.

## Funzionalità di base di enum

Come si è detto nel Volume 1, Capitolo 5, per visualizzare un elenco di costanti **enum** occorre chiamare il metodo **values()** di questo oggetto. Il metodo **values()** produce un array di costanti **enum** nell'ordine in cui sono state dichiarate consentendovi di utilizzare l'array risultante, per esempio in un ciclo **foreach**.

---

1. L'autore ringrazia Joshua Bloch per la sua collaborazione nella stesura di questo capitolo.

Quando create un'**enum**, il compilatore genera automaticamente una classe collegata; questa classe viene ereditata automaticamente da **java.lang.Enum** che fornisce alcune funzionalità, illustrate nell'esempio seguente:

```
//: enumerated/EnumClass.java
// Funzionalità della classe Enum
import static net.mindview.util.Print.*;

enum Shrubbery { GROUND, CRAWLING, HANGING }

public class EnumClass {
  public static void main(String[] args) {
    for(Shrubbery s : Shrubbery.values()) {
      print(s + " ordinal: " + s.ordinal());
      printnb(s.compareTo(Shrubbery.CRAWLING) + " ");
      printnb(s.equals(Shrubbery.CRAWLING) + " ");
      print(s == Shrubbery.CRAWLING);
      print(s.getDeclaringClass());
      print(s.name());
      print("----------------------");
    }
    // Produce un valore enum da un nome di stringa:
    for(String s : "HANGING CRAWLING GROUND".split(" ")) {
      Shrubbery shrub = Enum.valueOf(Shrubbery.class, s);
      print(shrub);
    }
  }
} /* Output:
GROUND ordinal: 0
-1 false false
class Shrubbery
GROUND
----------------------
CRAWLING ordinal: 1
0 true true
class Shrubbery
```

```
CRAWLING
----------------------
HANGING ordinal: 2
1 false false
class Shrubbery
HANGING
----------------------
HANGING
CRAWLING
GROUND
*///:~
```

Il metodo **ordinal()** restituisce un **int** che indica l'ordine di dichiarazione di ogni istanza **enum**, a partire da zero. Potete sempre confrontare le istanze di **enum** servendovi di **==**, e i metodi **equals()** e **hashCode()** verranno creati automaticamente. Essendo di tipo **Comparable**, la classe **Enum** dispone di un metodo **compareTo()** ed è anche serializzabile (**Serializable**).

Se chiamate **getDeclaringClass()** per un'istanza **enum**, otterrete la classe **enum** che la ingloba.

I metodi **name()** e **toString()** restituiscono il nome dell'**enum** esattamente come è stato dichiarato. Il metodo **valueOf()** è un membro **static** di **Enum** che fornisce l'istanza di **enum** corrispondente al nome di **String** passatogli, sollevando un'eccezione in caso di mancata corrispondenza.

## *Utilizzo delle importazioni static con le enum*

Considerate questa variante del programma **Burrito.java** (Volume 1, Capitolo 5):

```
//: enumerated/Spiciness.java
package enumerated;

public enum Spiciness {
  NOT, MILD, MEDIUM, HOT, FLAMING
} ///:~

//: enumerated/Burrito.java
package enumerated;
```

```
import static enumerated.Spiciness.*;

public class Burrito {
  Spiciness degree;
  public Burrito(Spiciness degree) { this.degree = degree;}
  public String toString() { return "Burrito is "+ degree;}
public static void main(String[] args) {
    System.out.println(new Burrito(NON_PICCANTE));
    System.out.println(new Burrito(MEDIAMENTE_PICCANTE));
    System.out.println(new Burrito(PICCANTE));
  }
} /* Output:
Burrito is NOT
Burrito is MEDIUM
Burrito is HOT
*///:~
```

Le istruzioni **import static** importano tutti gli identificativi di istanza **enum** nello spazio di nomi locale, in modo che non sia necessario qualificarli. Se si tratti di una tecnica valida o se sia preferibile essere espliciti e qualificare tutte le istanze di **enum** dipende unicamente dalla complessità del codice.

Il compilatore non vi consentirà di utilizzare il tipo errato, pertanto l'unica vostra preoccupazione sarà fare in modo che il codice non risulti confuso al lettore. Solitamente non avrete problemi, in ogni caso è opportuno che valutiate attentamente i singoli casi.

Tenete presente che non è possibile servirsi di questa tecnica se **enum** è definito nello stesso file o nel pacchetto predefinito.

## Aggiunta di metodi a un'enum

Se si esclude il fatto che non è permesso ereditare da essa, un'**enum** può essere gestita come una classe qualunque: questo significa che potete aggiungere dei metodi all'**enum** e persino dotarla di un metodo **main()**.

Da un'**enum** potete ottenere anche una descrizione diversa da quella fornita dal metodo predefinito **toString()**, che come si è detto produce soltanto il nome dell'istanza **enum**.

A questo scopo, potete fornire un costruttore per intercettare le informazioni supplementari e metodi aggiuntivi per ottenere una descrizione estesa, come nell'esempio seguente:

```
//: enumerated/OzWitch.java
// Le streghe del magico mondo di Oz.
import static net.mindview.util.Print.*;

public enum OzWitch {
  // Le istanze devono essere definite prima dei metodi:
  WEST("Miss Gulch, aka the Wicked Witch of the West"),
  NORTH("Glinda, the Good Witch of the North"),
  EAST("Wicked Witch of the East, wearer of the Ruby " +
    "Slippers, crushed by Dorothy's house"),
  SOUTH("Good by inference, but missing");
  private String description;
  // L'accesso del costruttore deve essere di tipo package
  // o private:
  private OzWitch(String description) {
    this.description = description;
  }
  public String getDescription() { return description; }
  public static void main(String[] args) {
    for(OzWitch witch : OzWitch.values())
      print(witch + ": " + witch.getDescription());
  }
} /* Output:
WEST: Miss Gulch, aka the Wicked Witch of the West
NORTH: Glinda, the Good Witch of the North
EAST: Wicked Witch of the East, wearer of the Ruby
Slippers, crushed by Dorothy's house
SOUTH: Good by inference, but missing
*///:~
```

Notate che, se state definendo i metodi, dovrete concludere la sequenza di istanze **enum** con un punto e virgola. Osservate inoltre che Java definisce per prima cosa le istanze in **enum**: se cercate di definirle dopo uno qualsiasi dei metodi o dei campi, otterrete un errore di compilazione.

Il costruttore e i metodi hanno formati identici a quelli di una normale classe perché, seppure con alcune limitazioni, **enum** *è* una classe normale a tutti gli effetti.

Nonostante il costruttore sia stato reso **private** in questo esempio, in realtà il tipo di accesso scelto non fa differenza, dal momento che il costruttore è utilizzabile soltanto per creare le istanze **enum** dichiarate all'interno della definizione **enum**: una volta che la definizione di **enum** è completa, il compilatore non vi consentirà di utilizzare il costruttore per creare nuove istanze.

## Sovrascrittura dei metodi di enum

Di seguito è presentata un'altra tecnica per ottenere diversi valori di stringa per i tipi enumerativi, mediante la quale i nomi delle istanze vengono semplicemente riformattati per la visualizzazione. La sovrascrittura del metodo **toString()** per un'**enum** è identica a quella di una classe qualunque:

```
//: enumerated/SpaceShip.java
public enum SpaceShip {
  SCOUT, CARGO, TRANSPORT, CRUISER, BATTLESHIP, MOTHERSHIP;
  public String toString() {
    String id = name();
    String lower = id.substring(1).toLowerCase();
    return id.charAt(0) + lower;
  }
  public static void main(String[] args) {
    for(SpaceShip s : values()) {
      System.out.println(s);
    }
  }
} /* Output:
Scout
Cargo
Transport
Cruiser
Battleship
Mothership
*///:~
```

Il metodo **toString()** ottiene il nome della **SpaceShip** chiamando il metodo **name()**, e ne trasforma in maiuscolo la lettera iniziale.

## Enum nelle dichiarazioni switch

Le **enum** risultano molto pratiche, se utilizzate all'interno di istruzioni **switch**. Di norma queste istruzioni funzionano soltanto con valori interi: tuttavia, poiché le **enum** hanno un ordine intero prestabilito e tenuto conto che l'ordine di un'istanza può essere ottenuto con il metodo **ordinal()**, le **enum** possono essere utilizzate nelle istruzioni **switch**.

Sebbene di norma un'istanza di **enum** debba essere qualificata dal suo tipo, non occorre che ciò avvenga in un'istruzione **case**. Considerate questo esempio, che si serve di un'**enum** per creare una semplice macchina a stati:

```
//: enumerated/TrafficLight.java
// Utilizzo delle Enum nelle dichiarazioni switch.
import static net.mindview.util.Print.*;

// Definizione di un tipo enum:
enum Signal { GREEN, YELLOW, RED, }

public class TrafficLight {
  Signal color = Signal.RED;
  public void change() {
    switch(color) {
      // Tenete presente che non occorre indicare Signal.RED
      // nella dichiarazione case:
      case RED:    color = Signal.GREEN;
                   break;
      case GREEN:  color = Signal.YELLOW;
                   break;
      case YELLOW: color = Signal.RED;
                   break;
    }
  }
  public String toString() {
    return "The traffic light is " + color;
```

```
  }
  public static void main(String[] args) {
    TrafficLight t = new TrafficLight();
    for(int i = 0; i < 7; i++) {
      print(t);
      t.change();
    }
  }
} /* Output:
The traffic light is RED
The traffic light is GREEN
The traffic light is YELLOW
The traffic light is RED
The traffic light is GREEN
The traffic light is YELLOW
The traffic light is RED
*///:~
```

Il compilatore non si cura dell'assenza di un'istruzione **default** all'interno di **switch**, ma questo non è dovuto al fatto che ogni istanza di **Signal** ha alcune dichiarazioni **case**: se commentaste una delle istruzioni **case**, il compilatore continuerebbe a non lamentarsi.

Questo significa che dovete prestare attenzione e accertarvi di avere tenuto in considerazione tutti i **case** possibili. D'altra parte, chiamando **return** dalle istruzioni **case** il compilatore *protesterà* se non trova il costrutto **default**, anche se nelle istruzioni **case** avrete tenuto conto di tutti i valori possibili di **enum**.

**Esercizio 1** (2) Servitevi di **static import** per modificare **TrafficLight.java** in modo da non dovere qualificare le istanze di **enum**.

## Il mistero di values()

Come si è detto, tutte le classi **enum** vengono create dal compilatore ed estendono la classe **Enum**; tuttavia, se analizzate **Enum**, noterete che non vi è alcun metodo **values()** anche se lo state utilizzando. Che esistano altri metodi "nascosti"? Per scoprirlo, potete scrivere un piccolo programma che si serve della riflessione:

```
//: enumerated/Reflection.java
```

```
// Analisi delle enum mediante la riflessione.
import java.lang.reflect.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

enum Explore { HERE, THERE }

public class Reflection {
  public static Set<String> analyze(Class<?> enumClass) {
    print("----- Analyzing " + enumClass + " -----");
    print("Interfaces:");
    for(Type t : enumClass.getGenericInterfaces())
      print(t);
    print("Base: " + enumClass.getSuperclass());
    print("Methods: ");
    Set<String> methods = new TreeSet<String>();
    for(Method m : enumClass.getMethods())
      methods.add(m.getName());
    print(methods);
    return methods;
  }
  public static void main(String[] args) {
    Set<String> exploreMethods = analyze(Explore.class);
    Set<String> enumMethods = analyze(Enum.class);
    print("Explore.containsAll(Enum)? " +
      exploreMethods.containsAll(enumMethods));
    printnb("Explore.removeAll(Enum): ");
    exploreMethods.removeAll(enumMethods);
    print(exploreMethods);
    // Decompila il codice dell'enum:
    OSExecute.command("javap Explore");
  }
} /* Output:
----- Analyzing class Explore -----
Interfaces:
Base: class java.lang.Enum
```

```
Methods:
[compareTo, equals, getClass, getDeclaringClass, hashCode,
name, notify, notifyAll, ordinal, toString, valueOf, values,
wait]
----- Analyzing class java.lang.Enum -----
Interfaces:
java.lang.Comparable<E>
interface java.io.Serializable
Base: class java.lang.Object
Methods:
[compareTo, equals, getClass, getDeclaringClass, hashCode,
name, notify, notifyAll, ordinal, toString, valueOf, wait]
Explore.containsAll(Enum)? true
Explore.removeAll(Enum): [values]
Compiled from "Reflection.java"
final class Explore extends java.lang.Enum{
    public static final Explore QUI;
    public static final Explore LA';
    public static final Explore[] values();
    public static Explore valueOf(java.lang.String);
    static {};
}
*///:~
```

La risposta è sì: **values()** è un metodo **static** aggiunto dal compilatore. Potete vedere che, nel corso della creazione di **enum**, anche **valueOf()** è aggiunto a **Explore**. Questa situazione è un po' confusa, dal momento che anche la classe **Enum** possiede un metodo **valueOf()**, la quale tuttavia ha due argomenti e non uno come il metodo che viene inserito. In ogni caso, il metodo **Set** utilizzato in questo esempio tiene conto dei soli nomi di metodo ignorando le segnature, pertanto dopo avere chiamato **Explore.removeAll(Enum)** tutto ciò che rimane è **[values]**.

L'output mostra che **Explore** è stato reso **final** dal compilatore, di conseguenza è impossibile ereditare da un'**enum**. Esiste anche una clausola di inizializzazione **static**, che come vedrete sarà ridefinita in seguito.

Per effetto della cancellazione (*erasure*) descritta nel Capitolo 3 di questo volume, il decompilatore non dispone di tutte le informazioni su **Enum**, quindi mostra la classe di base **Explore** come **Enum** grezza, anziché l'effettiva **Enum<Explore>**.

Poiché **values()** è un metodo **static** che il compilatore ha inserito nella definizione di **enum**, se eseguite l'upcast di un tipo **enum** a **Enum**, il metodo **values()** non sarà disponibile. Tenete presente, tuttavia, che **Class** dispone di un metodo **getEnumConstants()**, quindi anche se **values()** non appartiene all'interfaccia **Enum** potete ottenere le istanze di **enum** tramite l'oggetto **Class**:

```
//: enumerated/UpcastEnum.java
// Se si esegue l'upcast di un'enum il metodo values()
// non e' disponibile

enum Search { HITHER, YON }

public class UpcastEnum {
  public static void main(String[] args) {
    Search[] vals = Search.values();
    Enum e = Search.HITHER; // Upcast
    // e.values(); // Enum non ha values()
    for(Enum en : e.getClass().getEnumConstants())
      System.out.println(en);
  }
} /* Output:
HITHER
YON
*///:~
```

Visto che **getEnumConstants()** è un metodo **Class**, potete chiamarlo per una classe che non contiene tipi enumerativi:

```
//: enumerated/NonEnum.java

public class NonEnum {
  public static void main(String[] args) {
    Class<Integer> intClass = Integer.class;
    try {
      for(Object en : intClass.getEnumConstants())
        System.out.println(en);
    } catch(Exception e) {
      System.out.println(e);
```

```
    }
  }
} /* Output:
java.lang.NullPointerException
*///:~
```

Il metodo tuttavia restituisce **null**, quindi se provate a utilizzare il risultato otterrete un'eccezione.

## Implementazione, non ereditarietà

Ora sapete che tutte le **enum** estendono **java.lang.Enum**. Java non supporta l'ereditarietà multipla, di conseguenza non potete creare un'**enum** per ereditarietà:

```
enum NotPossible extends Pet { ... // Non funzionera'
```

È invece possibile creare un'**enum** che implementi una o più interfacce:

```
//: enumerated/cartoons/EnumImplementation.java
// Una enum puo' implementare un'interfaccia
package enumerated.cartoons;
import java.util.*;
import net.mindview.util.*;

enum CartoonCharacter
implements Generator<CartoonCharacter> {
  SLAPPY, SPANKY, PUNCHY, SILLY, BOUNCY, NUTTY, BOB;
  private Random rand = new Random(47);
  public CartoonCharacter next() {
    return values()[rand.nextInt(values().length)];
  }
}

public class EnumImplementation {
  public static <T> void printNext(Generator<T> rg) {
    System.out.print(rg.next() + ", ");
  }
```

```
  public static void main(String[] args) {
    // Sceglie un'istanza:
    CartoonCharacter cc = CartoonCharacter.BOB;
    for(int i = 0; i < 10; i++)
      printNext(cc);
  }
} /* Output:
BOB, PUNCHY, BOB, SPANKY, NUTTY, PUNCHY, SLAPPY, NUTTY, NUTTY,
SLAPPY,
*///:~
```

Il risultato è piuttosto strano, perché la chiamata di un metodo richiede la disponibilità di un'istanza di **enum**. Comunque ora l'**enum CartoonCharacter** può essere accettata da qualsiasi metodo che accetti un **Generator**, per esempio **printNext()**.

> **Esercizio 2** (2) Anziché implementare un'interfaccia, rendete **static** il metodo **next()**. Quali sono i benefici e gli inconvenienti di tale approccio?

## Selezione casuale

Molti esempi in questo capitolo richiedono la selezione tra istanze di **enum**, come avete visto in **CartoonCharacter.next()**. È possibile generalizzare questa operazione, utilizzando i generici, e inserire il risultato nella libreria comune **net.mindview.util**:

```
//: net/mindview/util/Enums.java
package net.mindview.util;
import java.util.*;

public class Enums {
  private static Random rand = new Random(47);
  public static <T extends Enum<T>> T random(Class<T> ec) {
    return random(ec.getEnumConstants());
  }
  public static <T> T random(T[] values) {
    return values[rand.nextInt(values.length)];
  }
```

```
} ///:~
```

La curiosa sintassi **<T extends Enum<T>>** descrive **T** come istanza di **enum**. Passando **Class<T>**, rendete disponibile l'oggetto **Class** e potete produrre l'array di istanze **enum**. Il metodo **random()** sovrascritto richiede soltanto di sapere che sta ottenendo **T[]** perché non deve eseguire le operazioni di **Enum**, ma semplicemente selezionare in modo casuale un elemento di array. Il tipo restituito è il tipo esatto dell'**enum**. Ecco un semplice test del metodo **random()**:

```
//: enumerated/RandomTest.java
import net.mindview.util.*;

enum Activity { SITTING, LYING, STANDING, HOPPING,
  RUNNING, DODGING, JUMPING, FALLING, FLYING }

public class RandomTest {
  public static void main(String[] args) {
    for(int i = 0; i < 20; i++)
      System.out.print(Enums.random(Activity.class) + " ");
  }
} /* Output:
STANDING FLYING RUNNING STANDING RUNNING STANDING LYING
DODGING SITTING RUNNING HOPPING HOPPING HOPPING RUNNING
STANDING LYING FALLING RUNNING FLYING LYING
*///:~
```

Anche se **Enums** è una classe relativamente piccola, come vedete impedisce una quantità ingente di duplicazione di codice in questo capitolo. La duplicazione genera errori, pertanto è sempre opportuno sforzarsi di eliminarla.

## Utilizzo delle interfacce per l'organizzazione

Spesso, l'impossibilità di ereditare da un'**enum** può rappresentare un inconveniente. Se fosse possibile ereditare da un'**enum**, infatti, si potrebbe estendere il numero di elementi originali, nonché creare alcune sottocategorie ricorrendo ai sottotipi.

Potete ottenere questo genere di categorizzazione raggruppando gli elementi all'interno di un'interfaccia e creando un'**enum** basata su quell'interfaccia. Per esempio, supponete di avere diverse classi di alimenti che vorreste creare come **enum**, pur volendo che ogni alimento continui a essere un tipo di **Food**. Ecco un possibile risultato:

```
//: enumerated/menu/Food.java
// Classificazione di enum in sottocategorie nell'ambito
// delle interfacce.
package enumerated.menu;

public interface Food {
  enum Appetizer implements Food {
    SALAD, SOUP, SPRING_ROLLS;
  }
  enum MainCourse implements Food {
    LASAGNE, BURRITO, PAD_THAI,
    LENTILS, HUMMOUS, VINDALOO;
  }
  enum Dessert implements Food {
    TIRAMISU, GELATO, BLACK_FOREST_CAKE,
    FRUIT, CREME_CARAMEL;
  }
  enum Coffee implements Food {
    BLACK_COFFEE, DECAF_COFFEE, ESPRESSO,
    LATTE, CAPPUCCINO, TEA, HERB_TEA;
  }
} ///:~
```

Poiché l'unica sottotipizzazione disponibile per un'**enum** è quella fornita dall'implementazione dell'interfaccia, ogni **enum** nidificata implementa l'interfaccia **Food** che la incorpora. Come vedete, ora è possibile affermare che "tutto è un tipo di **Food**":

```
//: enumerated/menu/TypeOfFood.java
package enumerated.menu;
import static enumerated.menu.Food.*;
```

```
public class TypeOfFood {
  public static void main(String[] args) {
    Food food = Appetizer.SALAD;
    food = MainCourse.LASAGNE;
    food = Dessert.GELATO;
    food = Coffee.CAPPUCCINO;
  }
} ///:~
```

L'upcast a **Food** funziona per ogni tipo di **enum** che implementa (**implements**) **Food**, pertanto tutti sono tipi di **Food**.

Un'interfaccia, tuttavia, non è utile come **enum** quando occorra gestire un insieme di tipi. Se desiderate "un'**enum** di **enum**" potete creare un'**enum** che la incorpora, con un'istanza per ogni **enum** in **Food**:

```
//: enumerated/menu/Course.java
package enumerated.menu;
import net.mindview.util.*;

public enum Course {
  APPETIZER(Food.Appetizer.class),
  MAINCOURSE(Food.MainCourse.class),
  DESSERT(Food.Dessert.class),
  COFFEE(Food.Coffee.class);
  private Food[] values;
  private Course(Class<? extends Food> kind) {
    values = kind.getEnumConstants();
  }
  public Food randomSelection() {
    return Enums.random(values);
  }
} ///:~
```

Ciascuna di queste **enum** accetta l'oggetto **Class** corrispondente come argomento del costruttore, da cui può estrarre e memorizzare tutte le istanze **enum** servendosi di **getEnumConstants()**.

In seguito tali istanze vengono usate in **randomSelection()**, in modo da creare un menu casuale di piatti con la selezione di un elemento **Food** da ogni **Course**:

```
//: enumerated/menu/Meal.java
package enumerated.menu;

public class Meal {
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      for(Course course : Course.values()) {
        Food food = course.randomSelection();
        System.out.println(food);
      }
      System.out.println("---");
    }
  }
} /* Output:
SPRING_ROLLS
VINDALOO
FRUIT
DECAF_COFFEE
---
SOUP
VINDALOO
FRUIT
TEA
---
SALAD
BURRITO
FRUIT
TEA
---
SALAD
BURRITO
CREME_CARAMEL
MILK
```

```
---
SOUP
BURRITO
TIRAMISU
ESPRESSO
---
*///:~
```

In questo caso, il vantaggio ottenuto dalla creazione di **enum** di **enum** è la possibilità di iterare ogni **Course**. Nel prosieguo del capitolo, nell'esempio **VendingMachine.java** vedrete un altro approccio alla categorizzazione che è imposta da vincoli differenti.

Un altro meccanismo di categorizzazione, più compatto, consiste nel nidificare le **enum** all'interno di altre **enum**, nel modo seguente:

```
//: enumerated/SecurityCategory.java
// Variante più compatta della sottocategorizzazione di enum.
import net.mindview.util.*;

enum SecurityCategory {
  STOCK(Security.Stock.class), BOND(Security.Bond.class);
  Security[] values;
  SecurityCategory(Class<? extends Security> kind) {
    values = kind.getEnumConstants();
  }
  interface Security {
    enum Stock implements Security { SHORT, LONG, MARGIN }
    enum Bond implements Security { MUNICIPAL, JUNK }
  }
  public Security randomSelection() {
    return Enums.random(values);
  }
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++) {
      SecurityCategory category =
        Enums.random(SecurityCategory.class);
      System.out.println(category + ": " +
```

```
        category.randomSelection());
    }
  }
} /* Output:
BOND: MUNICIPAL
BOND: MUNICIPAL
STOCK: MARGIN
STOCK: MARGIN
BOND: JUNK
STOCK: SHORT
STOCK: LONG
STOCK: LONG
BOND: MUNICIPAL
BOND: JUNK
*///:~
```

L'interfaccia **Security** è necessaria per raccogliere in un tipo comune le **enum** contenute, che vengono poi categorizzate nelle **enum** all'interno di **SecurityCategory**.

Se adottate la stessa tecnica per l'esempio di **Food**, il risultato sarà il seguente:

```
//: enumerated/menu/Meal2.java
package enumerated.menu;
import net.mindview.util.*;

public enum Meal2 {
  APPETIZER(Food.Appetizer.class),
  MAINCOURSE(Food.MainCourse.class),
  DESSERT(Food.Dessert.class),
  COFFEE(Food.Coffee.class);
  private Food[] values;
  private Meal2(Class<? extends Food> kind) {
    values = kind.getEnumConstants();
  }
  public interface Food {
    enum Appetizer implements Food {
```

```
      SALAD, SOUP, SPRING_ROLLS;
    }
    enum MainCourse implements Food {
      LASAGNE, BURRITO, PAD_THAI,
      LENTILS, HUMMOUS, VINDALOO;
    }
    enum Dessert implements Food {
      TIRAMISU, ICE-CREAM, BLACK_FOREST_CAKE,
      FRUIT, CREME_CARAMEL;
    }
    enum Coffee implements Food {
      BLACK_COFFEE, DECAF_COFFEE, ESPRESSO,
      MILK, CAPPUCCINO, TEA, HERB_TEA;
    }
  }
  public Food randomSelection() {
    return Enums.random(values);
  }
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      for(Meal2 meal : Meal2.values()) {
        Food food = meal.randomSelection();
        System.out.println(food);
      }
      System.out.println("---");
    }
  }
} /* Stesso output di Meal.java *///:~
```

Alla fine, si tratta soltanto di una riorganizzazione del codice, che però in alcuni casi può produrre una struttura molto più lineare.

**Esercizio 3** (1) Aggiungete un nuovo **Course** a **Course.java** e dimostratene il funzionamento in **Meal.java**.

**Esercizio 4** (1) Ripetete lo stesso esercizio per **Meal2.java**.

**Esercizio 5** (4) Modificate **control/VowelsAndConsonants.java** in modo che utilizzi tre tipi di **enum**: **VOWEL** per le vocali, **SOMETIMES_A_VOWEL** per le semivocali (j e y) e **CONSONANT** per le consonanti.

Il costruttore di **enum** dovrebbe accettare le varie lettere che descrivono quella categoria particolare. Suggerimento: utilizzate gli argomenti variabili (vararg) e ricordatevi che questi creano automaticamente gli array.

**Esercizio 6** (3) Verificate se si ottiene un vantaggio dalla nidificazione di **Appetizer**, **MainCourse**, **Dessert** e **Coffee** all'interno di **Food**, in alternativa alla loro trasformazione in **enum** autonome che implementano **Food**.

## Utilizzo di EnumSet in alternativa ai flag

Un **Set** è un tipo di collezione che consente l'aggiunta di un solo oggetto per ogni tipo. Ovviamente **enum** richiede che tutti i membri siano univoci, in modo da emulare il normale comportamento di un insieme **Set**, ma dato che non è permesso aggiungere né rimuovere gli elementi, in fin dei conti l'insieme non risulta molto utile.

La classe **EnumSet** è stata aggiunta in Java SE5 per operare di concerto con le **enum**, in alternativa ai tradizionali "bit flag" basati su **int**. Questi flag sono utilizzati per indicare un tipo di informazioni di tipo binario (on/off, acceso/spento, 0/1 ecc.), ma alla fine vi ritrovate a manipolare bit invece di concetti, e questo rende confuso il vostro codice.

La classe **EnumSet** è stata progettata per competere efficacemente in velocità con i bit flag, garantendo operazioni di norma molto più rapide rispetto a un **HashSet**. Internamente **EnumSet** è rappresentata (se possibile) da un unico **long** trattato come vettore di bit, pertanto è molto veloce ed efficiente. Il vantaggio è che in questo modo potete disporre di un metodo molto più espressivo per indicare la presenza o l'assenza di una funzionalità binaria, senza dovervi preoccupare delle prestazioni.

Gli elementi di un **EnumSet** devono provenire da una sola **enum**. Considerate questo esempio, che utilizza un'**enum** per indicare le posizioni in cui sono disposti i sensori d'allarme in un palazzo:

```
//: enumerated/AlarmPoints.java
package enumerated;
public enum AlarmPoints {
  STAIR1, STAIR2, LOBBY, OFFICE1, OFFICE2, OFFICE3,
  OFFICE4, BATHROOM, UTILITY, KITCHEN
} ///:~
```

Per tenere traccia dello stato degli allarmi può essere utilizzato **EnumSet**:

```
//: enumerated/EnumSets.java
// Operazioni sugli EnumSet
package enumerated;
import java.util.*;
import static enumerated.AlarmPoints.*;
import static net.mindview.util.Print.*;

public class EnumSets {
  public static void main(String[] args) {
    EnumSet<AlarmPoints> points =
      EnumSet.noneOf(AlarmPoints.class); // Insieme vuoto
    points.add(BATHROOM);
    print(points);
    points.addAll(EnumSet.of(STAIR1, STAIR2, KITCHEN));
    print(points);
    points = EnumSet.allOf(AlarmPoints.class);
    points.removeAll(EnumSet.of(STAIR1, STAIR2, KITCHEN));
    print(points);
    points.removeAll(EnumSet.range(OFFICE1, OFFICE4));
    print(points);
    points = EnumSet.complementOf(points);
    print(points);
  }
} /* Output:
[BATHROOM]
[STAIR1, STAIR2, BATHROOM, KITCHEN]
[LOBBY, OFFICE1, OFFICE2, OFFICE3, OFFICE4, BATHROOM,
UTILITY]
[LOBBY, BATHROOM, UTILITY]
[STAIR1, STAIR2, OFFICE1, OFFICE2, OFFICE3, OFFICE4,
KITCHEN]
*///:~
```

Lo scopo delle dichiarazioni **static import** è facilitare l'utilizzo delle costanti **enum**. I nomi di metodo sono abbastanza autoesplicativi, e comunque accuratamente descritti nella documentazione JDK. Quando esaminerete questa documenta-

zione, rileverete un particolare interessante: il metodo **of()** è stato sovraccaricato sia con argomenti variabili (vararg) sia con singoli metodi che accettano da due a cinque argomenti espliciti. Questo dimostra l'importanza delle prestazioni di **EnumSet**: un solo metodo **of()** abbinato agli argomenti variabili avrebbe risolto il problema, ma in modo meno efficiente dell'utilizzo di argomenti espliciti. Quindi, se chiamate **of()** con un numero di argomenti da due a cinque, otterrete chiamate di metodo esplicite più rapide, mentre se chiamate **of()** con un solo argomento o più di cinque utilizzerete la versione di **of()** con i vararg. Ricordate che eseguendo la chiamata con un solo argomento il compilatore non costruirà l'array di vararg, quindi non incorrerete in oneri supplementari.

Gli **EnumSet** sono costituiti da **long**, ciascuno dei quali equivale a 64 bit, e ogni istanza di **enum** richiede un bit per indicare la situazione di presenza o assenza: questo significa che potete avere un **EnumSet** per un'**enum** di 64 elementi al massimo, senza eccedere la capienza di un valore **long**. Ma che cosa accade se l'**enum** contiene più di 64 elementi?

```
//: enumerated/BigEnumSet.java
import java.util.*;

public class BigEnumSet {
  enum Big { A0, A1, A2, A3, A4, A5, A6, A7, A8, A9, A10,
    A11, A12, A13, A14, A15, A16, A17, A18, A19, A20, A21,
    A22, A23, A24, A25, A26, A27, A28, A29, A30, A31, A32,
    A33, A34, A35, A36, A37, A38, A39, A40, A41, A42, A43,
    A44, A45, A46, A47, A48, A49, A50, A51, A52, A53, A54,
    A55, A56, A57, A58, A59, A60, A61, A62, A63, A64, A65,
    A66, A67, A68, A69, A70, A71, A72, A73, A74, A75 }
  public static void main(String[] args) {
    EnumSet<Big> bigEnumSet = EnumSet.allOf(Big.class);
    System.out.println(bigEnumSet);
  }
} /* Output:
[A0, A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, A13,
A14, A15, A16, A17, A18, A19, A20, A21, A22, A23, A24, A25, A26,
A27, A28, A29, A30, A31, A32, A33, A34, A35, A36, A37, A38, A39,
A40, A41, A42, A43, A44, A45, A46, A47, A48, A49, A50, A51, A52,
A53, A54, A55, A56, A57, A58, A59, A60, A61, A62, A63, A64, A65,
A66, A67, A68, A69, A70, A71, A72, A73, A74, A75]
*///:~
```

Dall'output risulta evidente che **EnumSet** non ha alcun problema con un'**enum** di oltre 64 elementi, quindi è ragionevole supporre che all'occorrenza Java aggiunga automaticamente un altro **long**.

**Esercizio 7** (3) Trovate il codice sorgente per **EnumSet** e illustratene il funzionamento.

## Utilizzo di EnumMap

Un'**EnumMap** è una **Map** speciale nella quale le chiavi devono far parte della stessa **enum**. Tenuto conto dei vincoli delle **enum**, le **EnumMap** possono essere implementate internamente come array e questo le rende molto veloci, permettendovi di utilizzare senza problemi le **EnumMap** per ricerche basate su **enum**.

Potete chiamare **put()** soltanto per le chiavi che sono presenti nell'**enum**, ma per tutto il resto è come se utilizzaste una **Map** standard.

L'esempio seguente dimostra l'utilizzo del design pattern *Command*. Questo modello inizia con un'interfaccia contenente di solito un solo metodo e crea diverse implementazioni con comportamenti differenti per il metodo in questione. Installate oggetti *Command* e il vostro programma li chiamerà quando necessario:

```
//: enumerated/EnumMaps.java
// Funzionalità di base delle EnumMap.
package enumerated;
import java.util.*;
import static enumerated.AlarmPoints.*;
import static net.mindview.util.Print.*;

interface Command { void action(); }

public class EnumMaps {
  public static void main(String[] args) {
    EnumMap<AlarmPoints,Command> em =
      new EnumMap<AlarmPoints,Command>(AlarmPoints.class);
    em.put(KITCKEN, new Command() {
      public void action() { print("Kitchen fire!"); }
    });
```

```
    em.put(BATHROOM, new Command() {
      public void action() { print("Bathroom alert!"); }
    });
    for(Map.Entry<AlarmPoints,Command> e : em.entrySet()) {
      printnb(e.getKey() + ": ");
      e.getValue().action();
    }
    try { // In assenza di valore per una determinata chiave:
      em.get(UTILITY).action();
    } catch(Exception e) {
      print(e);
    }
  }
} /* Output:
BATHROOM: Bathroom alert!
KITCHEN: Kitchen fire!
java.lang.NullPointerException
*///:~
```

Esattamente come avviene per un **EnumSet**, l'ordine degli elementi nella **EnumMap** è determinato dal loro ordine di definizione nell'**enum**.

L'ultima parte di **main()** indica che ogni **enum** ha sempre una voce di chiave, il cui valore è **null** a meno che non abbiate chiamato **put()** per quella chiave.

Un vantaggio di un'**EnumMap** rispetto ai *metodi specifici per le costanti*, descritti di seguito, è che l'**EnumMap** consente di modificare il valore degli oggetti, mentre vedrete che per i metodi specifici per le costanti tale valore è fissato al momento della compilazione.

Come vedrete nel prosieguo di capitolo, le **EnumMap** possono essere impiegate per implementare la "trasmissione multipla" (*multiple dispatching*) nelle situazioni in cui tipi multipli di **enum** devono poter interagire reciprocamente.

## Metodi specifici per le costanti

Le **enum** di Java presentano una caratteristica molto interessante che vi consente di assegnare a ogni loro istanza un comportamento differente, creando metodi per ciascuna istanza. Per fare questo occorre definire uno o più meto-

di **abstract** come componente di **enum**, poi definire i metodi per ogni istanza di **enum**. Per esempio:

```
//: enumerated/ConstantSpecificMethod.java
import java.util.*;
import java.text.*;

public enum ConstantSpecificMethod {
  DATE_TIME {
    String getInfo() {
      return
        DateFormat.getDateInstance().format(new Date());
    }
  },
  CLASSPATH {
    String getInfo() {
      return System.getenv("CLASSPATH");
    }
  },
  VERSION {
    String getInfo() {
      return System.getProperty("java.version");
    }
  };
  abstract String getInfo();
  public static void main(String[] args) {
    for(ConstantSpecificMethod csm : values())
      System.out.println(csm.getInfo());
  }
} /* (Da eseguire per visualizzare l'output) *///:~
```

Potete interrogare e chiamare i metodi mediante l'istanza di **enum** a essi associata. Questa tecnica, che presenta alcune analogie con il modello Command, è spesso denominata *table-driven code*, letteralmente "codice azionato dalle tabelle".

Nella programmazione OOP, comportamenti diversi sono associati a classi differenti. Poiché ogni istanza di **enum** può avere un comportamento auto-

nomo, dettato dai metodi specifici per le costanti, questo suggerisce che ogni caso sia un tipo da considerare separatamente. Nell'esempio precedente, ogni caso di **enum** è trattato come il "tipo di base" **ConstantSpecificMethod**, dal quale si può ottenere un comportamento polimorfico chiamando il metodo **getInfo()**.

Dovete tuttavia prendere questa analogia per quello che è, poiché non è possibile gestire le istanze di **enum** come tipi di classe:

```
//: enumerated/NotClasses.java
// {Exec: javap -c LikeClasses}
import static net.mindview.util.Print.*;

enum LikeClasses {
  WINKEN { void behavior() { print("Behavior1"); } },
  BLINKEN { void behavior() { print("Behavior2"); } },
  NOD { void behavior() { print("Behavior3"); } };
  abstract void behavior();
}

public class NotClasses {
  // void f1(LikeClasses.WINKEN instance) {} // No
} /* Output:
Compiled from "NotClasses.java"
abstract class LikeClasses extends java.lang.Enum{
public static final LikeClasses WINKEN;

public static final LikeClasses BLINKEN;

public static final LikeClasses NOD;
...
*///:~
```

In **f1()** potete osservare che il compilatore non consente l'utilizzo di un'istanza **enum** come tipo di classe, il che ha senso se considerate il codice generato dal compilatore, in cui ogni elemento di **enum** è un'istanza **static final** di **LikeClasses**.

Inoltre, essendo **static**, le istanze **enum** delle **enum** interne non si comportano come classi interne normali: non potete accedere ai campi o ai metodi non **static** della classe esterna.

Un esempio più interessante è quello di un autolavaggio che offre ai clienti una serie di opzioni, ciascuna delle quali corrisponde a un'azione diversa. È possibile associare a ogni opzione un metodo specifico per le costanti, mentre l'**EnumSet** può registrare le selezioni operate dal cliente:

```
//: enumerated/CarWash.java
import java.util.*;
import static net.mindview.util.Print.*;

public class CarWash {
  public enum Cycle {
    UNDERBODY {
      void action() { print("Spraying the underbody"); }
    },
    WHEELWASH {
      void action() { print("Washing the wheels"); }
    },
    PREWASH {
     void action() { print("Loosening the dirt"); }
    },
    BASIC {
     void action() { print("The basic wash"); }
    },
    HOTWAX {
     void action() { print("Applying hot wax"); }
    },
    RINSE {
     void action() { print("Rinsing"); }
    },
    BLOWDRY {
     void action() { print("Blowing dry");
    }
  abstract void action();
  }
```

```
  EnumSet<Cycle> cycles =
     EnumSet.of(Cycle.BASIC, Cycle.RINSE);
  public void add(Cycle cycle) { cycles.add(cycle); }
  public void washCar() {
    for(Cycle c : cycles)
      c.action();
  }
  public String toString() { return cycles.toString(); }
  public static void main(String[] args) {
    CarWash wash = new CarWash();
    print(wash);
    wash.washCar();
    // L'ordine di aggiunta e' ininfluente:
    wash.add(Cycle.BLOWDRY);
    wash.add(Cycle.BLOWDRY); // Duplicates ignored
    wash.add(Cycle.RINSE);
    wash.add(Cycle.HOTWAX);
    print(wash);
    wash.washCar();
  }
} /* Output:
[BASIC, RINSE]
The basic wash
Rinsing
[BASIC, HOTWAX, RINSE, BLOWDRY]
The basic wash
Applying hot wax
Rinsing
Blowing dry
*///:~
```

La sintassi per la definizione di un metodo specifico per le costanti è quella di una classe interna anonima, ma più concisa.

Questo esempio mostra altre caratteristiche degli **EnumSet**. Trattandosi di un insieme, conterrà soltanto un elemento per ogni voce, pertanto le chiamate ad **add()** che hanno lo stesso argomento verranno ignorate: questo è perfettamente logico, dal momento che è possibile attivare un bit su "on" una volta

soltanto. Inoltre, l'ordine di aggiunta delle istanze di **enum** è ininfluente, poiché l'ordine di output è determinato da quello di dichiarazione dell'**enum**.

Come potete vedere di seguito, anziché implementare un metodo **abstract**, è possibile sovrascrivere i metodi specifici per le costanti:

```
//: enumerated/OverrideConstantSpecific.java
import static net.mindview.util.Print.*;

public enum OverrideConstantSpecific {
  NUT, BOLT,
  WASHER {
    void f() { print("Overridden method"); }
  };
  void f() { print("default behavior"); }
  public static void main(String[] args) {
    for(OverrideConstantSpecific ocs : values()) {
      printnb(ocs + ": ");
      ocs.f();
    }
  }
} /* Output:
NUT: default behavior
BOLT: default behavior
WASHER: Overridden method
*///:~
```

Ricordate che sebbene le **enum** impediscano la scrittura di determinate forme di codice, generalmente dovreste testare questi oggetti come se fossero classi.

## *Catena di responsabilità con le enum*

Nel design pattern *Chain of Responsibility* per risolvere un problema vengono creati un certo numero di metodi diversi, che poi sono concatenati. Ogni richiesta in arrivo viene passata alla catena, finché arriva alla specifica soluzione che è in grado di gestirla.

Potete implementare facilmente una "catena di responsabilità" con i metodi specifici per le costanti. Considerate un modello di ufficio postale, che cerca di gestire ogni missiva nel modo più generale possibile, ma che prova un cer-

to numero di invii prima di considerare la missiva come "non inoltrabile". Ogni tentativo può essere visto come una "strategia" (pattern *Strategy*) e l'intera lista come una *Chain of Responsibility*.

Iniziate con la descrizione di una missiva le cui caratteristiche possono essere espresse per mezzo di **enum**. Poiché gli oggetti **Mail** saranno generati a caso, il modo più facile per ridurre la probabilità, per esempio, che a una parte di posta sia assegnato l'indicatore **YES** per **GeneralDelivery** è creare più istanze di non **YES**, pertanto le definizioni di **enum** vi appariranno alquanto insolite.

All'interno di **Mail**, il metodo **randomMail()** genera alcune missive casuali. Il metodo **generator()** produce un oggetto **Iterable** che utilizza **randomMail()** per creare un certo numero di missive, una per ogni chiamata a **next()** tramite iteratore. Questo costrutto consente la semplice realizzazione di un ciclo **foreach**, chiamando **Mail.generator()**:

```
//: enumerated/PostOffice.java
// Modellizzazione di un ufficio postale.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

class Mail {
  // I NO riducono la probabilità di selezione casuale:
  enum GeneralDelivery {YES,NO1,NO2,NO3,NO4,NO5}
  enum Scannability {UNSCANNABLE,YES1,YES2,YES3,YES4}
  enum Readability {ILLEGIBLE,YES1,YES2,YES3,YES4}
  enum Address {INCORRECT,OK1,OK2,OK3,OK4,OK5,OK6}
  enum ReturnAddress {MISSING,OK1,OK2,OK3,OK4,OK5}
  GeneralDelivery generalDelivery;
  Scannability scannability;
  Readability readability;
  Address address;
  ReturnAddress returnAddress;
  static long counter = 0;
  long id = counter++;
  public String toString() { return "Mail " + id; }
  public String details() {
    return toString() +
```

```
      ", General Delivery: " + generalDelivery +
      ", Address Scanability: " + scannability +
      ", Address Readability: " + readability +
      ", Address Address: " + address +
      ", Return address: " + returnAddress;
  }
  // Generazione missive di prova:
  public static Mail randomMail() {
    Mail m = new Mail();
    m.generalDelivery= Enums.random(GeneralDelivery.class);
    m.scannability = Enums.random(Scannability.class);
    m.readability = Enums.random(Readability.class);
    m.address = Enums.random(Address.class);
    m.returnAddress = Enums.random(ReturnAddress.class);
    return m;
  }
  public static Iterable<Mail> generator(final int count) {
    return new Iterable<Mail>() {
      int n = count;
      public Iterator<Mail> iterator() {
        return new Iterator<Mail>() {
          public boolean hasNext() { return n-- > 0; }
          public Mail next() { return randomMail(); }
          public void remove() { // Non implementato
            throw new UnsupportedOperationException();
          }
        };
      }
    };
  }
}

public class PostOffice {
  enum MailHandler {
    GENERAL_DELIVERY {
      boolean handle(Mail m) {
```

```
      switch(m.generalDelivery) {
        case YES:
          print("Using general delivery for " + m);
          return true;
        default: return false;
      }
    }
  },
  MACHINE_SCAN {
    boolean handle(Mail m) {
      switch(m.scannability) {
        case UNSCANNABLE: return false;
        default:
          switch(m.address) {
            case INCORRECT: return false;
            default:
              print("Delivering "+ m + " automatically");
              return true;
          }
      }
    }
  },
  VISUAL_INSPECTION {
    boolean handle(Mail m) {
      switch(m.readability) {
        case ILLEGIBLE: return false;
        default:
          switch(m.address) {
            case INCORRECT: return false;
            default:
              print("Delivering " + m + " normally");
              return true;
          }
      }
    }
  },
```

```
    RETURN_TO_SENDER {
      boolean handle(Mail m) {
        switch(m.returnAddress) {
          case MISSING: return false;
          default:
            print("Returning " + m + " to sender");
            return true;
        }
      }
    };
    abstract boolean handle(Mail m);
  }
  static void handle(Mail m) {
    for(MailHandler handler : MailHandler.values())
      if(handler.handle(m))
        return;
    print(m + " is a dead letter");
  }
  public static void main(String[] args) {
    for(Mail mail : Mail.generator(10)) {
      print(mail.details());
      handle(mail);
      print("*****");
    }
  }
} /* Output:
Mail 0, General Delivery: NO2, Address Scanability:
UNSCANNABLE, Address Readability: YES3, Address Address:
OK1, Return address: OK1
Delivering Mail 0 normally
*****
Mail 1, General Delivery: NO5, Address Scanability: YES3,
Address Readability: ILLEGIBLE, Address Address: OK5,
Return address: OK1
Delivering Mail 1 automatically
*****
Mail 2, General Delivery: YES, Address Scanability: YES3,
```

```
Address Readability: YES1, Address Address: OK1, Return
address: OK5
Using general delivery for Mail 2
*****
Mail 3, General Delivery: NO4, Address Scanability: YES3,
Address Readability: YES1, Address Address: INCORRECT,
Return address: OK4
Returning Mail 3 to sender
*****
Mail 4, General Delivery: NO4, Address Scanability:
UNSCANNABLE, Address Readability: YES1, Address Address:
INCORRECT, Return address: OK2
Returning Mail 4 to sender
*****
Mail 5, General Delivery: NO3, Address Scanability: YES1,
Address Readability: ILLEGIBLE, Address Address: OK4,
Return address: OK2
Delivering Mail 5 automatically
*****
Mail 6, General Delivery: YES, Address Scanability: YES4,
Address Readability: ILLEGIBLE, Address Address: OK4,
Return address: OK4
Using general delivery for Mail 6
*****
Mail 7, General Delivery: YES, Address Scanability: YES3,
Address Readability: YES4, Address Address: OK2, Return
address: MISSING
Using general delivery for Mail 7
*****
Mail 8, General Delivery: NO3, Address Scanability: YES1,
Address Readability: YES3, Address Address: INCORRECT,
Return address: MISSING
Mail 8 is a dead letter
*****
Mail 9, General Delivery: NO1, Address Scanability:
UNSCANNABLE, Address Readability: YES2, Address Address:
OK1, Return address: OK4
```

```
Delivering Mail 9 normally
*****
*///:~
```

La catena di responsabilità è espressa in **enum MailHandler**, e l'ordine delle definizioni di **enum** determina l'ordine in cui le strategie vengono eseguite a fronte di ogni missiva. Ogni strategia viene provata finché non se ne trovi una che abbia successo; se tutte falliscono, la missiva dovrà essere considerata come non inoltrabile.

**Esercizio 8** (6) Modificate **PostOffice.java** in modo che possa inviare la posta.

**Esercizio 9** (5) Modificate la classe **PostOffice** in modo che utilizzi **EnumMap**.

**Progetto** I linguaggi specializzati come Prolog utilizzano il *concatenamento inverso* (*backward chaining*) per risolvere problemi analoghi. Ispirandovi a **PostOffice.java**, analizzate questi linguaggi e sviluppate un programma che consenta di aggiungere facilmente nuove "regole" al sistema.[2]

## Macchine a stati con enum

Come si è detto, i tipi enumerativi sono ideali per la creazione delle *macchine a stati*. Questa espressione definisce un sistema che può trovarsi in un numero limitato di condizioni specifiche; normalmente la macchina si muove da una condizione all'altra in base a un input, ma esistono anche le cosiddette *condizioni transitorie*, dalle quali la macchina si sposta non appena la loro attività è terminata.

Ogni stato ammette un certo numero di input, e input diversi portano la macchina a una condizione diversa. Poiché le **enum** limitano l'insieme di casi possibili, sono particolarmente utili per enumerare i vari stati e input.

Inoltre, di solito ogni stato produce un determinato output.

Un distributore automatico è un buon esempio di macchina a stati. In primo luogo, definite i vari input in un'**enum**:

```
//: enumerated/Input.java
package enumerated;
```

2. I progetti sono proposte da utilizzare, per esempio, come pianificazioni a termine. Le soluzioni ai progetti non sono incluse nella guida delle soluzioni agli esercizi.

```
import java.util.*;

public enum Input {
  NICKEL(5), DIME(10), QUARTER(25), DOLLAR(100),
  TOOTHPASTE(200), CHIPS(75), SODA(100), SOAP(50),
  ABORT_TRANSACTION {
    public int amount() { // Non permesso
      throw new RuntimeException("ABORT.amount()");
    }
  },
  STOP { // Questa deve essere l'ultima istanza.
    public int amount() { // Non permesso
      throw new RuntimeException("SHUT_DOWN.amount()");
    }
  };
  int value; // In centesimi
  Input(int value) { this.value = value; }
  Input() {}
  int amount() { return value; }; // In centesimi
  static Random rand = new Random(47);
  public static Input randomSelection() {
    // Non include STOP:
    return values()[rand.nextInt(values().length - 1)];
  }
} ///:~
```

Notate che due serie di **Input** sono associate a un importo (quelli sulla riga che inizia con NICKEL(5) e quelli sulla riga di TOOTHPASTE(200)), per cui **amount()** deve essere definito nell'interfaccia. Non si può tuttavia consentire la chiamata al metodo **amount()** per gli altri due tipi di **Input** (ABORT_TRANSACTION e STOP): nel caso in cui avvenga la chiamata ad **amount()**, verrà sollevata un'eccezione. Definire un metodo in un'interfaccia e poi sollevare eccezioni per determinate implementazioni di tale metodo è certamente una tecnica piuttosto insolita, ma è resa necessaria dai vincoli delle **enum**.

La **VendingMachine** reagirà a questi input innanzitutto categorizzandoli tramite l'**enum Category**, in modo da poter eseguire **switch** sulle categorie.

L'esempio seguente mostra come le **enum** contribuiscano a rendere il codice più chiaro e facile da gestire:

```
//: enumerated/VendingMachine.java
// {Args: VendingMachineInput.txt}
package enumerated;
import java.util.*;
import net.mindview.util.*;
import static enumerated.Input.*;
import static net.mindview.util.Print.*;

enum Category {
  MONEY(NICKEL, DIME, QUARTER, DOLLAR),
  ITEM_SELECTION(TOOTHPASTE, CHIPS, SODA, SOAP),
  QUIT_TRANSACTION(ABORT_TRANSACTION),
  SHUT_DOWN(STOP);
  private Input[] values;
  Category(Input... types) { values = types; }
  private static EnumMap<Input,Category> categories =
    new EnumMap<Input,Category>(Input.class);
  static {
    for(Category c : Category.class.getEnumConstants())
      for(Input type : c.values)
        categories.put(type, c);
  }
  public static Category categorize(Input input) {
    return categories.get(input);
  }
}

public class VendingMachine {
  private static State state = State.RESTING;
  private static int amount = 0;
  private static Input selection = null;
  enum StateDuration { TRANSIENT } // Collega enum alle
                                   // categorie di stato
```

```
enum State {
  RESTING {
    void next(Input input) {
      switch(Category.categorize(input)) {
        case MONEY:
          amount += input.amount();
          state = ADDING_MONEY;
          break;
        case SHUT_DOWN:
          state = TERMINAL;
        default:
      }
    }
  },
  ADDING_MONEY {
    void next(Input input) {
      switch(Category.categorize(input)) {
        case MONEY:
          amount += input.amount();
          break;
        case ITEM_SELECTION:
          selection = input;
          if(amount < selection.amount())
            print("Insufficient money for " + selection);
          else state = DISPENSING;
          break;
        case QUIT_TRANSACTION:
          state = GIVING_CHANGE;
          break;
        case SHUT_DOWN:
          state = TERMINAL;
        default:
      }
    }
  },
  DISPENSING(StateDuration.TRANSIENT) {
    void next() {
```

```
      print("here is your " + selection);
      amount -= selection.amount();
      state = GIVING_CHANGE;
    }
  },
  GIVING_CHANGE(StateDuration.TRANSIENT) {
    void next() {
      if(amount > 0) {
        print("Your change: " + amount);
        amount = 0;
      }
      state = RESTING;
    }
  },
  TERMINAL { void output() { print("Halted"); } };
  private boolean isTransient = false;
  State() {}
  State(StateDuration trans) { isTransient = true; }
  void next(Input input) {
    throw new RuntimeException("Only call " +
      "next(Input input) for non-transient states");
  }
  void next() {
    throw new RuntimeException("Only call next() for " +
      "StateDuration.TRANSIENT states");
  }
  void output() { print(amount); }
}
static void run(Generator<Input> gen) {
  while(state != State.TERMINAL) {
    state.next(gen.next());
    while(state.isTransient)
      state.next();
    state.output();
  }
}
```

```
  public static void main(String[] args) {
    Generator<Input> gen = new RandomInputGenerator();
    if(args.length == 1)
      gen = new FileInputGenerator(args[0]);
    run(gen);
  }
}

// Per un controllo di base:
class RandomInputGenerator implements Generator<Input> {
  public Input next() { return Input.randomSelection(); }
}

// Crea gli Input da un file di stringhe separate da ';':
class FileInputGenerator implements Generator<Input> {
  private Iterator<String> input;
  public FileInputGenerator(String fileName) {
    input = new TextFile(fileName, ";").iterator();
  }
  public Input next() {
    if(!input.hasNext())
      return null;
    return Enum.valueOf(Input.class, input.next().trim());
  }
} /* Output:
25
50
75
here is your CHIPS
0
100
200
here is your TOOTHPASTE
0
25
35
Your change: 35
```

```
0
25
35
Insufficient money for SODA
35
60
70
75
Insufficient money for SODA
75
Your change: 75
0
Halted
*///:~
```

Dal momento che la selezione dei diversi casi di **enum** avviene spesso per mezzo di un'istruzione **switch**, una delle domande che dovreste porvi quando state organizzando **enum** multiple è: "Come creare le necessarie corrispondenze con l'istruzione **switch**?". In questo caso specifico, è più facile procedere a ritroso all'analisi della **VendingMachine**. Considerate che in ogni **State** è necessario sottoporre a **switch** le categorie principali di azioni di input: inserimento delle monete (MONEY), selezione di un prodotto (ITEM_SELECTION), interruzione della transazione (QUIT_TRANSACTION) e spegnimento del distributore automatico (SHUT_DOWN). Queste categorie, tuttavia, prevedono al loro interno diversi tipi di monete e di articoli selezionabili. L'**enum Category** raggruppa i vari tipi di **Input** in modo che il metodo **categorize()** possa produrre la **Category** adatta all'interno di uno **switch**. Questo metodo si serve di una **EnumMap** per eseguire la ricerca in modo efficiente e sicuro.

Analizzando la classe **VendingMachine**, noterete che ogni condizione è diversa e risponde all'input in modo diverso. Notate anche i due stati transitori, GIVING_CHANGE e DISPENSING. In **run()** la macchina aspetta un **Input** e non smette di spostarsi attraverso gli stati fino a quando non esce da una condizione transitoria.

Il codice di **VendingMachine** può essere testato in due modi, utilizzando due oggetti **Generator** diversi. **RandomInputGenerator** continua a produrre gli input, tranne **SHUT_DOWN**; l'esecuzione continua di questo generatore esegue una sorta di test per verificare che la macchina non entri in una condizione "difettosa". Il generatore **FileInputGenerator**, invece, accetta

un file di testo con la descrizione degli input, li trasforma in istanze di **enum** e crea gli oggetti di **Input**. Ecco il file utilizzato per produrre l'output precedente:

```
//:! enumerated/VendingMachineInput.txt
QUARTER; QUARTER; QUARTER; CHIPS;
DOLLAR; DOLLAR; TOOTHPASTE;
QUARTER; DIME; ABORT_TRANSACTION;
QUARTER; DIME; SODA;
QUARTER; DIME; NICKEL; SODA;
ABORT_TRANSACTION;
STOP;
///:~
```

Una limitazione di questo progetto è che i campi di **VendingMachine** cui hanno accesso le istanze **State** dell'**enum** *devono* essere **static**, il che implica che è possibile avere una sola istanza di **VendingMachine**. Tenete presente, in ogni caso, che questo potrebbe non essere un problema in un'effettiva implementazione Java nativa, dal momento che ogni distributore difficilmente avrà più di un'applicazione di questo tipo.

**Esercizio 10** (7) Modificate la sola classe **VendingMachine** utilizzando **EnumMap**, in modo che un programma possa avere istanze multiple di **VendingMachine**.

**Esercizio 11** (7) In un distributore automatico è ragionevole pensare che sia possibile aggiungere e modificare facilmente il tipo di articoli venduti, pertanto i limiti imposti da un'**enum** a **Input** sono poco pratici: ricordate che le **enum** gestiscono un insieme limitato di tipi. Modificate **VendingMachine.java** in modo che gli articoli venduti siano rappresentati da una **class** anziché essere parte di **Input**, e inizializzate un **ArrayList** di questi oggetti a partire da un file di testo, servendovi di **net.mindview.util.TextFile**.

**Progetto** Progettate il distributore automatico in un'ottica di internazionalizzazione, in modo che la macchina possa essere adattata alle diverse lingue nazionali.[3]

3. I progetti sono proposte da utilizzare, per esempio, come pianificazioni a termine. Le soluzioni ai progetti non sono incluse nella guida delle soluzioni agli esercizi.

## Multiple dispatching

Quando dovete gestire diversi tipi d'interazione, il vostro programma rischia di diventare molto complesso. Per esempio, in un sistema che analizza ed esegue espressioni matematiche vorrete esprimere **Number.plus (Number)**, **Number.multiply (Number)** ecc., dove **Number** è la classe di base per una famiglia di oggetti numerici. Ma quando dite **a.plus(b)** e non conoscete il tipo esatto di **a** opure di **b**, come potrete farli interagire in modo corretto?

Il dilemma ha a che fare con qualcosa a cui probabilmente non avete pensato, ovvero il fatto che Java esegue soltanto il cosiddetto *single dispatching*: in pratica, se eseguite un'operazione su oggetti multipli dei quali non conoscete il tipo di appartenenza, Java potrà attivare il meccanismo di polimorfismo (Volume 1, Capitolo 8) soltanto a fronte di uno di questi tipi. Tale comportamento non risolve il problema descritto, di conseguenza finirete per rilevare alcuni tipi manualmente e riprodurre voi stessi la funzionalità di polimorfismo che vi occorre.

La soluzione è chiamata *multiple dispatching*, letteralmente "trasmissione multipla"; nel caso specifico, che richiede soltanto due invii di messaggi, si utilizza il termine *double dispatching*. Il polimorfismo può verificarsi soltanto tramite le chiamate di metodo, pertanto se desiderate un *double dispatching*, devono esistere due chiamate di metodo: la prima per determinare il primo tipo sconosciuto e la seconda per determinare il secondo. Con il multiple dispatching dovete prevedere una chiamata virtuale per ciascun tipo: se lavorate con due diverse gerarchie di tipi che devono interagire, vi occorrerà una chiamata virtuale in ogni gerarchia. Di norma, imposterete una configurazione tale che una singola chiamata di metodo produca più chiamate di metodo virtuali e quindi serva più tipi all'interno del processo. Per ottenere questo effetto, dovrete lavorare con più metodi e avrete bisogno di una chiamata di metodo per ogni invio di messaggio. I metodi utilizzati nell'esempio seguente, che implementa il gioco della morra (*RoShamBo*), sono **compete()** ed **eval()**, entrambi membri dello stesso tipo. Questi due metodi producono tre risultati possibili:[4]

```
//: enumerated/Outcome.java
package enumerated;
public enum Outcome { WIN, LOSE, DRAW } ///:~
```

4. Dopo essere stato pubblicato per alcuni anni su *www.mindview.net* sia nella versione in C++ sia in quella Java (in *Thinking in Patterns*), questo esempio è apparso, senza attribuzione, in un libro di altri autori.

```
//: enumerated/RoShamBo1.java
// Dimostrazione di "multiple dispatching".
package enumerated;
import java.util.*;
import static enumerated.Outcome.*;

interface Item {
  Outcome compete(Item it);
  Outcome eval(Paper p);
  Outcome eval(Scissors s);
  Outcome eval(Rock r);
}

class Paper implements Item {
  public Outcome compete(Item it) { return it.eval(this); }
  public Outcome eval(Paper p) { return DRAW; }
  public Outcome eval(Scissors s) { return WIN; }
  public Outcome eval(Rock r) { return LOSE; }
  public String toString() { return "Paper"; }
}

class Scissors implements Item {
  public Outcome compete(Item it) { return it.eval(this); }
  public Outcome eval(Paper p) { return LOSE; }
  public Outcome eval(Scissors s) { return DRAW; }
  public Outcome eval(Rock r) { return WIN; }
  public String toString() { return "Scissors"; }
}

class Rock implements Item {
  public Outcome compete(Item it) { return it.eval(this); }
  public Outcome eval(Paper p) { return WIN; }
  public Outcome eval(Scissors s) { return LOSE; }
  public Outcome eval(Rock r) { return DRAW; }
  public String toString() { return "Rock"; }
}
```

```
public class RoShamBo1 {
  static final int SIZE = 20;
  private static Random rand = new Random(47);
  public static Item newItem() {
    switch(rand.nextInt(3)) {
      default:
      case 0: return new Scissors();
      case 1: return new Paper();
      case 2: return new Rock();
    }
  }
  public static void match(Item a, Item b) {
    System.out.println(
      a + " vs. " + b + ": " +  a.compete(b));
  }
  public static void main(String[] args) {
    for(int i = 0; i < SIZE; i++)
      match(newItem(), newItem());
  }
} /* Output:
Rock vs. Rock: DRAW
Paper vs. Rock: WIN
Paper vs. Rock: WIN
Paper vs. Rock: WIN
Scissors vs. Paper: WIN
Scissors vs. Scissors: DRAW
Scissors vs. Paper: WIN
Rock vs. Paper: LOSE
Paper vs. Paper: DRAW
Rock vs. Paper: LOSE
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
Rock vs. Scissors: WIN
Rock vs. Paper: LOSE
Paper vs. Rock: WIN
Scissors vs. Paper: WIN
```

```
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
*///:~
```

**Item** è l'interfaccia per i tipi che saranno oggetto di *multiple dispatching*. **RoShamBo1.match()** accetta due oggetti **Item** e inizia il processo di "doppio invio", chiamando la funzione **Item.compete()**. Il meccanismo virtuale determina il tipo **a**, pertanto si attiva all'interno della funzione **compete()** del tipo concreto **a**.

La funzione **compete()** esegue il secondo invio chiamando **eval()** sul tipo rimasto. Passando se stesso (**this**) come argomento a **eval()**, produce una chiamata alla funzione **eval()** sovraccarica, preservando così le informazioni di tipo del primo invio. Al termine del secondo invio, quindi, conoscerete i tipi esatti di entrambi gli oggetti **Item**.

L'impostazione *multiple dispatching* richiede sicuramente alcuni accorgimenti, ma non dimenticate quale vantaggio in termini di eleganza sintattica potrete ottenere al momento della chiamata: invece di scrivere codice intricato per determinare il tipo di uno o più oggetti durante la chiamata, non dovrete fare altro che dire: "Non mi interessa di che tipo siete, voglio che interagiate a vicenda in modo corretto!".

## *Dispatching con enum*

Realizzare la "traduzione" diretta di **RoShamBo1.java** in una soluzione basata su **enum** è un compito problematico, poiché le istanze di **enum** non sono tipi, e di conseguenza i metodi **eval()** sovraccarichi non funzioneranno: non è possibile utilizzare le istanze **enum** come tipi di argomento. Esistono tuttavia diversi approcci al *multiple dispatching* che traggono beneficio dalle **enum**.

Una tecnica si serve di un costruttore per inizializzare ogni istanza **enum** con una "riga" di risultati; questo processo consente di ottenere una sorta di tabella di ricerca:

```
//: enumerated/RoShamBo2.java
// Come passare da un'enum a un'altra.
package enumerated;
import static enumerated.Outcome.*;

public enum RoShamBo2 implements Competitor<RoShamBo2> {
```

```
  PAPER(DRAW, LOSE, WIN),
  SCISSORS(WIN, DRAW, LOSE),
  ROCK(LOSE, WIN, DRAW);
  private Outcome vPAPER, vSCISSORS, vROCK;
  RoShamBo2(Outcome paper,Outcome scissors,Outcome rock) {
    this.vPAPER = paper;
    this.vSCISSORS = scissors;
    this.vROCK = rock;
  }
  public Outcome compete(RoShamBo2 it) {
    switch(it) {
      default:
      case PAPER: return vPAPER;
      case SCISSORS: return vSCISSORS;
      case ROCK: return vROCK;
    }
  }
  public static void main(String[] args) {
    RoShamBo.play(RoShamBo2.class, 20);
  }
} /* Output:
ROCK vs. ROCK: DRAW
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
PAPER vs. PAPER: DRAW
PAPER vs. SCISSORS: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. SCISSORS: DRAW
ROCK vs. SCISSORS: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
ROCK vs. PAPER: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. ROCK: LOSE
```

```
PAPER vs. SCISSORS: LOSE
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
*///:~
```

Una volta che in **compete()** sono stati determinati entrambi i tipi, l'unica azione eseguita è la restituzione dell'**Outcome** risultante. Nulla vi vieta, tuttavia, di chiamare un altro metodo, anche, per esempio, tramite un oggetto *Command* che sia stato assegnato nel costruttore.

Il codice di **RoShamBo2.java** è molto più ridotto e immediato dell'esempio originale, quindi è più semplice mantenerlo aggiornato. Notate che state utilizzando due invii per determinare il tipo di entrambi gli oggetti. In **RoShamBo1.java** entrambi gli invii sono stati eseguiti mediante chiamate di metodo virtuali, ma in questo caso soltanto il primo invio ricorre a una chiamata virtuale: il secondo invio, infatti, utilizza **switch**, tuttavia è sicuro perché **enum** limita le scelte nell'istruzione **switch**.

Il codice che "pilota" l'**enum** è stato scorporato, in modo che sia possibile utilizzarlo negli altri esempi. Per prima cosa, l'interfaccia **Competitor** definisce un tipo che è in competizione con un altro **Competitor**:

```
//: enumerated/Competitor.java
// Come passare da un'enum a un'altra.
package enumerated;

public interface Competitor<T extends Competitor<T>> {
  Outcome compete(T competitor);
} ///:~
```

Quindi vengono definiti due metodi **static**: questa parola chiave evita di specificare esplicitamente il tipo di parametro. In primo luogo, **match()** chiama **compete()** per mettere a confronto i **Competitor**; notate che in questo caso il parametro di tipo deve soltanto essere **Competitor<T>**. In **play()**, tuttavia, il parametro di tipo deve essere sia un **Enum<T>**, perché utilizzato in **Enums.random()**, sia un **Competitor<T>**, perché passato a **match()**:

```
//: enumerated/RoShamBo.java
// Strumenti comuni per gli esempi RoShamBo.
```

```
package enumerated;
import net.mindview.util.*;

public class RoShamBo {
  public static <T extends Competitor<T>>
  void match(T a, T b) {
    System.out.println(
      a + " vs. " + b + ": " +  a.compete(b));
  }
  public static <T extends Enum<T> & Competitor<T>>
  void play(Class<T> rsbClass, int size) {
    for(int i = 0; i < size; i++)
      match(
        Enums.random(rsbClass),Enums.random(rsbClass));
  }
} ///:~
```

Il metodo **play()** non ha un valore di ritorno che coinvolga il parametro di tipo **T**: a prima vista, quindi, sembrerebbe possibile l'utilizzo di metacaratteri all'interno del tipo **Class<T>**, invece della descrizione del parametro principale. I metacaratteri tuttavia non possono estendere più di un tipo di base, pertanto è necessario utilizzare l'espressione sopra riportata.

## Utilizzo dei metodi specifici per le costanti

Poiché i metodi specifici per le costanti consentono di fornire implementazioni di metodo diverse per ogni istanza **enum**, potrebbero sembrare la soluzione perfetta per impostare il *multiple dispatching*. Anche se in questo modo si possono ottenere diversi comportamenti, le istanze di **enum** non sono tipi, quindi non possono essere utilizzati come tipi di argomento nelle segnature di metodo. Il meglio che possiate fare in questo esempio è impostare un'istruzione **switch**:

```
//: enumerated/RoShamBo3.java
// Utilizzo di metodi specifici per le costanti.
package enumerated;
import static enumerated.Outcome.*;

public enum RoShamBo3 implements Competitor<RoShamBo3> {
```

```
  PAPER {
    public Outcome compete(RoShamBo3 it) {
      switch(it) {
        default: // Per "tenere a bada" il compilatore
        case PAPER: return DRAW;
        case SCISSORS: return LOSE;
        case ROCK: return WIN;
      }
    }
  },
  SCISSORS {
    public Outcome compete(RoShamBo3 it) {
      switch(it) {
        default:
        case PAPER: return WIN;
        case SCISSORS: return DRAW;
        case ROCK: return LOSE;
      }
    }
  },
  ROCK {
    public Outcome compete(RoShamBo3 it) {
      switch(it) {
        default:
        case PAPER: return LOSE;
        case SCISSORS: return WIN;
        case ROCK: return DRAW;
      }
    }
  };
  public abstract Outcome compete(RoShamBo3 it);
  public static void main(String[] args) {
    RoShamBo.play(RoShamBo3.class, 20);
  }
} /* Stesso output di RoShamBo2.java
*///:~
```

Anche se il codice di questo esempio è funzionale, la precedente soluzione **RoShamBo2.java** richiede una quantità inferiore di codice per l'aggiunta di un nuovo tipo, quindi sembra la soluzione più pratica.

**RoShamBo3.java** può essere tuttavia semplificato e compattato:

```
//: enumerated/RoShamBo4.java
package enumerated;

public enum RoShamBo4 implements Competitor<RoShamBo4> {
  ROCK {
    public Outcome compete(RoShamBo4 opponent) {
      return compete(SCISSORS, opponent);
    }
  },
  SCISSORS {
    public Outcome compete(RoShamBo4 opponent) {
      return compete(PAPER, opponent);
    }
  },
  PAPER {
    public Outcome compete(RoShamBo4 opponent) {
      return compete(ROCK, opponent);
    }
  };
  Outcome compete(RoShamBo4 loser, RoShamBo4 opponent) {
    return ((opponent == this) ? Outcome.DRAW
        : ((opponent == loser) ? Outcome.WIN
                               : Outcome.LOSE));
  }
  public static void main(String[] args) {
    RoShamBo.play(RoShamBo4.class, 20);
  }
} /* Stesso output di RoShamBo2.java *///:~
```

In questo codice, il secondo invio viene eseguito dalla versione a due argomenti di **compete()**, che mette in opera una sequenza di confronti ed è quindi simile all'azione di uno **switch**. Tenete presente che, pur es-

sendo più compatta, questa soluzione risulta un po' confusa e di conseguenza potrebbe non essere idonea per applicazioni o sistemi di grandi dimensioni.

## *Dispatching con le EnumMap*

È possibile realizzare un "vero" *double dispatching* servendosi della classe **EnumMap**, che è stata concepita specificamente per funzionare in modo efficiente con le **enum**. Considerato che l'obiettivo è scegliere tra due tipi sconosciuti, potete utilizzare un'**EnumMap** di **EnumMap** per realizzare il *double dispatching*:

```
//: enumerated/RoShamBo5.java
// Multiple dispatching che utilizza una EnumMap di EnumMap.
package enumerated;
import java.util.*;
import static enumerated.Outcome.*;

enum RoShamBo5 implements Competitor<RoShamBo5> {
  PAPER, SCISSORS, ROCK;
  static EnumMap<RoShamBo5,EnumMap<RoShamBo5,Outcome>>
    table = new EnumMap<RoShamBo5,
      EnumMap<RoShamBo5,Outcome>>(RoShamBo5.class);
  static {
    for(RoShamBo5 it : RoShamBo5.values())
      table.put(it,
        new EnumMap<RoShamBo5,Outcome>(RoShamBo5.class));
    initRow(PAPER, DRAW, LOSE, WIN);
    initRow(SCISSORS, WIN, DRAW, LOSE);
    initRow(ROCK, LOSE, WIN, DRAW);
  }
  static void initRow(RoShamBo5 it,
    Outcome vPAPER, Outcome vSCISSORS, Outcome vROCK) {
    EnumMap<RoShamBo5,Outcome> row =
      RoShamBo5.table.get(it);
    row.put(RoShamBo5.PAPER, vPAPER);
    row.put(RoShamBo5.SCISSORS, vSCISSORS);
    row.put(RoShamBo5.ROCK, vROCK);
```

```
  }
  public Outcome compete(RoShamBo5 it) {
    return table.get(this).get(it);
  }
  public static void main(String[] args) {
    RoShamBo.play(RoShamBo5.class, 20);
  }
} /* Stesso output di RoShamBo2.java *///:~
```

La classe **EnumMap** è inizializzata mediante una clausola **static**; osservate la struttura, simile a una tabella, per le chiamate a **initRow()**. Notate anche il metodo **compete()**, in cui entrambi i *dispatching* avvengono in una sola istruzione.

## *Utilizzo di un array bidimensionale*

Potete semplificare ancora di più la soluzione considerando che ogni istanza di **enum** ha un valore fisso, basato sul suo ordine di dichiarazione, e che **ordinal()** produce questo valore. Un array bidimensionale che fa corrispondere i competitori ai risultati offre la soluzione più concisa, diretta e probabilmente più veloce, pur tenendo conto che **EnumMap** utilizza un array interno:

```
//: enumerated/RoShamBo6.java
// Enum che utilizzano delle "tabelle" invece del multiple
// dispatching.
package enumerated;
import static enumerated.Outcome.*;

enum RoShamBo6 implements Competitor<RoShamBo6> {
  PAPER, SCISSORS, ROCK;
  private static Outcome[][] table = {
    { DRAW, LOSE, WIN }, // PAPER
    { WIN, DRAW, LOSE }, // SCISSORS
    { LOSE, WIN, DRAW }, // ROCK
  };
  public Outcome compete(RoShamBo6 other) {
    return table[this.ordinal()][other.ordinal()];
```

```
  }
  public static void main(String[] args) {
    RoShamBo.play(RoShamBo6.class, 20);
  }
} ///:~
```

L'array **table** ha esattamente lo stesso ordine delle chiamate a **initRow()** nell'esempio precedente.

Le dimensioni ridotte di questo codice rappresentano un notevole vantaggio rispetto agli esempi precedenti, in parte perché è molto più facile da capire e modificare e in parte anche perché sembra più immediato; l'esempio non è tuttavia così "sicuro" come gli altri, perché ricorre a un array. Array più grandi potrebbero darvi problemi di dimensioni, e se i test che eseguirete non copriranno tutte le possibilità, rischierete di incorrere in altri inconvenienti.

Tutte queste soluzioni sono basate su tipi di tabelle diversi; vale comunque la pena fare alcuni esperimenti per trovare la forma che meglio garantisce la necessaria espressività del codice. Tenete presente che la soluzione precedente, pur essendo la più compatta, è anche estremamente rigida, perché produce un output costante a fronte di input costanti. In ogni caso, nulla vieta che **table** restituisca un oggetto funzione; per determinati tipi di problemi, infatti, il concetto di codice "pilotato" da una tabella può rivelarsi molto potente.

## Riepilogo

Sebbene i tipi enumerativi non siano complessi in sé, si è ritenuto di posporre questo capitolo in considerazione delle ampie possibilità offerte dalle **enum** in abbinamento a funzionalità quali il polimorfismo, i generici e la riflessione.

Anche se sono ben più sofisticate dei loro omonimi in C e C++, le **enum** rimangono una "piccola" funzionalità, qualcosa di cui Java ha fatto a meno, seppure con qualche difficoltà, per molti anni. In questo capitolo avete tuttavia visto quali effetti una funzionalità anche "piccola" può avere sul linguaggio, che talvolta vi offre lo strumento ideale per risolvere un problema in modo chiaro ed elegante. E, come si è detto più volte, l'eleganza è importante e la chiarezza può fare la differenza tra un programma ben riuscito e uno che non ha successo perché gli altri programmatori non sono in grado di comprenderlo.

Per rimanere in tema, una certa confusione è derivata dalla scelta infelice che in Java 1.0 ha portato a utilizzare il termine "enumeration" (anziché il

termine più comune e ormai accettato di “iterator”) per indicare un oggetto che seleziona ogni elemento di una sequenza, come avete visto a proposito delle Collection. Questa scelta è stata poi rivista in Java ma rimane nell'interfaccia **Enumeration**, la quale tuttavia non può essere semplicemente rimossa per ragioni di compatibilità con il vecchio codice, i riferimenti di libreria e la documentazione.

*La soluzione degli esercizi è disponibile nel documento The Thinking in Java Annotated Solution Guide, in vendita all'indirizzo www.mindview.net.*

# Capitolo 8

# Annotazioni

Le annotazioni, note anche come metadati, sono un modo formale per aggiungere al vostro codice informazioni che siano facilmente utilizzabili in un momento successivo.[1]

Le annotazioni sono motivate parzialmente da una generale tendenza dei linguaggi all'integrazione dei metadati nei file del codice sorgente, invece di mantenere queste informazioni in documenti esterni, ma sono allo stesso tempo una risposta alle nuove funzionalità offerte da altri linguaggi, come C#. Le annotazioni sono uno dei cambiamenti fondamentali introdotti nel linguaggio da Java SE5: forniscono le informazioni che vi occorrono per descrivere completamente il vostro programma, ma che non possono essere espresse in Java, e consentono di memorizzare informazioni supplementari in un formato che viene esaminato e verificato dal compilatore. Le annotazioni possono essere utilizzate per generare file descrittori (*descriptor file*) e persino nuove definizioni di classe, e semplificano l'onere di scrivere codice "riciclabile". Grazie alle annotazioni potete mantenere questi metadati nel codice sorgente di Java e usufruire di numerosi vantaggi: codice più chiaro, controllo in fase di compilazione e API specifiche per sviluppare

1. Jeremy Meyer ha dedicato due intere settimane a lavorare con l'autore a questo capitolo: il suo aiuto è stato inestimabile.

strumenti di elaborazione delle annotazioni. Sebbene alcuni tipi di metadati siano predefiniti in Java SE5, la decisione sul tipo di annotazioni che aggiungerete e il loro scopo sono interamente di vostra competenza.

La sintassi delle annotazioni è molto semplice ed è costituita essenzialmente dall'aggiunta del simbolo @ al linguaggio. Java SE5 offre tre annotazioni già incorporate, definite in **java.lang**, destinate a un utilizzo generico.

1. **@Override**, per indicare che una definizione di metodo dovrà sovrascrivere un metodo della classe di base; genera un errore di compilazione qualora l'ortografia del nome del metodo o la segnatura non siano corrette.[2]
2. **@Deprecated**, che produce un avvertimento del compilatore se viene utilizzato l'elemento corrente.
3. **@SuppressWarnings**, per disattivare gli avvertimenti inadeguati del compilatore. Questa annotazione è permessa ma non è supportata nelle versioni precedenti Java SE5, che la ignorano.

Come vedrete nel prosieguo del capitolo, altri quattro tipi di annotazione supportano la creazione.

Ogniqualvolta si creano classi o interfacce di descrittore che prevedono attività ripetute è possibile ricorrere alle annotazioni per automatizzare e facilitare il processo. La maggior parte dell'attività supplementare in *Enterprise JavaBeans* (EJB), per esempio, viene eliminata ricorrendo alle annotazioni in EJB 3.0.

Le annotazioni possono sostituire i sistemi esistenti, per esempio XDoclet, un software indipendente (si veda il supplemento *http://MindView.net/Books/BetterJava*) progettato per la creazione di doclet nello stile delle annotazioni.

Le annotazioni sono costrutti nativi del linguaggio e come tali strutturate e sottoposte a controllo di tipo al momento della compilazione. Conservando tutte le informazioni nel codice sorgente e non nei commenti, il codice risulterà più accurato e più facile da mantenere.

Utilizzando ed estendendo le API e gli strumenti per le annotazioni, e con il supporto di librerie esterne per la manipolazione del bytecode, che vedrete in questo capitolo, potete controllare e manipolare sia il vostro codice sorgente sia il bytecode.

2. Senza dubbio questa annotazione è stata ispirata da un'analoga funzionalità di C#, che tuttavia è una parola chiave, non un'annotazione, ed è imposta dal compilatore: infatti, quando sovrascrivete un metodo in C# dovete utilizzare la parola chiave **override**, mentre in Java l'annotazione **@Override** è facoltativa.

## Sintassi di base

Nell'esempio che segue il metodo **testExecute()** è annotato con **@Test**; in sé questo non produce nulla, ma il compilatore si accerterà che il percorso di compilazione preveda una definizione per l'annotazione **@Test**. Come vedrete nel prosieguo del capitolo, è possibile creare uno strumento che esegue questo metodo automaticamente tramite la riflessione.

```
//: annotations/Testable.java
package annotations;
import net.mindview.atunit.*;

public class Testable {
  public void execute() {
    System.out.println("Executing...");
  }
  @Test void testExecute() { execute(); }
} ///:~
```

I metodi annotati sono differenti dagli altri. L'annotazione **@Test** di questo esempio può essere utilizzata in abbinamento a modificatori quali **public**, **static** o **void**. Dal punto di vista sintattico l'utilizzo delle annotazioni è molto simile a quello dei modificatori.

### *Definizione delle annotazioni*

Questa è la definizione dell'annotazione elencata in precedenza. Potete notare come le definizioni di annotazione ricordino da vicino quelle di un'interfaccia; in effetti si compilano come file di classe, al pari di qualunque altra interfaccia di Java.

```
//: net/mindview/atunit/Test.java
// Il tag @Test.
package net.mindview.atunit;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {} ///:~
```

Se si esclude il simbolo @, la definizione **@Test** è del tutto analoga a quella di un'interfaccia vuota. La definizione di un'annotazione richiede anche le *meta-annotazioni* **@Target** e **@Retention**: **@Target** definisce (in **ElementType**) dove applicare l'annotazione corrente (a un metodo o a un campo, per esempio), mentre **@Retention** definisce (in **RetentionPolicy**) se le annotazioni sono disponibili nel codice sorgente (**SOURCE**), nel file di classe (**CLASS**), oppure durante l'esecuzione (**RUNTIME**).

Di norma le annotazioni contengono elementi che ne specificano i valori; un programma o uno strumento potranno servirsi di questi parametri al momento di elaborare le vostre annotazioni. Gli elementi sono simili ai metodi dell'interfaccia, ma consentono di dichiarare valori predefiniti.

Un'annotazione priva di elementi, come la precedente **@Test**, è denominata "annotazione marker".

Considerate questa semplice annotazione che tiene traccia dei casi d'utilizzo in un progetto, grazie alla quale i programmatori annotano ogni metodo o insieme di metodi che soddisfano i requisiti di uno specifico caso d'utilizzo. Un manager di progetto può così verificare lo stato di avanzamento del progetto contando i casi d'utilizzo implementati, e gli sviluppatori che lavorano al progetto possono facilmente trovarli, qualora debbano aggiornare o mettere a punto la logica applicativa.

```
//: annotations/UseCase.java
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface UseCase {
  public int id();
  public String description() default "no description";
} ///:~
```

Notate che **id** e **description** ricordano le dichiarazioni di metodo. Dal momento che **id** viene controllato dal compilatore per quanto riguarda i tipi, rappresenta un sistema affidabile per collegare un database di rilevamenti al documento di caso d'utilizzo e al codice sorgente.

L'elemento **description** ha un valore predefinito (**default**) che viene preso in carico dall'elaboratore delle annotazioni (*annotation processor*) qualora al momento dell'annotazione del metodo non sia stato specificato un altro valore.

La classe seguente ha tre metodi annotati come casi di utilizzo.

```
//: annotations/PasswordUtils.java
import java.util.*;

public class PasswordUtils {
  @UseCase(id = 47, description =
  "Passwords must contain at least one numeric")
  public boolean validatePassword(String password) {
    return (password.matches("\\w*\\d\\w*"));
  }
  @UseCase(id = 48)
  public String encryptPassword(String password) {
   return new StringBuilder(password).reverse().toString();
  }
  @UseCase(id = 49, description =
  "New passwords can't equal previously used ones")
  public boolean checkForNewPassword(
    List<String> prevPasswords, String password) {
    return !prevPasswords.contains(password);
  }
} ///:~
```

I valori degli elementi di annotazione sono coppie di nome-valore espresse in parentesi subito dopo la dichiarazione **@UseCase**. All'annotazione **encryptPassword()** non viene passato un valore per l'elemento **description**, pertanto il valore predefinito presente in **@interface UseCase** apparirà quando la classe verrà eseguita da un elaboratore di annotazioni.

Potreste immaginare di utilizzare un sistema di questo tipo per "abbozzare" la vostra applicazione, che riempirete via via con le funzionalità che svilupperete.

## Meta-annotazioni

Attualmente Java definisce soltanto tre annotazioni standard, che vedrete tra poco, e quattro meta-annotazioni. Le meta-annotazioni servono per annotare annotazioni.

| | |
|---|---|
| **@Target** | Indica dove può essere applicata l'applicazione. Gli argomenti possibili di **ElementType** sono i seguenti:<br>**CONSTRUCTOR**: dichiarazione di costruttore;<br>**FIELD**: dichiarazione di campo (include le costanti **enum**);<br>**LOCAL_VARIABLE**: dichiarazione di variabile locale;<br>**METHOD**: dichiarazione di metodo;<br>**PACKAGE**: dichiarazione di pacchetto;<br>**PARAMETER**: dichiarazione di parametro;<br>**TYPE**: dichiarazione di classe, interfaccia (incluso il tipo di annotazione), o di **enum**. |
| **@Retention** | Indica per quanto tempo vengono mantenute le informazioni di annotazione. Gli argomenti possibili di **RetentionPolicy** sono:<br>**SOURCE**: le annotazioni sono scartate dal compilatore;<br>**CLASS**: le annotazioni vengono registrate nel file di classe dal compilatore, tuttavia non devono necessariamente essere conservate dalla JVM durante l'esecuzione;<br>**RUNTIME**: le annotazioni sono conservate dalla JVM durante l'esecuzione, quindi possono essere lette tramite la riflessione. |
| **@Documented** | Include l'annotazione in Javadoc. |
| **@Inherited** | Consente alle sottoclassi di ereditare le annotazioni dal genitore. |

Nella maggior parte dei casi, definirete le vostre annotazioni e scriverete gli elaboratori che le gestiscono.

## Scrivere elaboratori di annotazioni

Senza strumenti che le leggano, difficilmente le annotazioni vi saranno più utili dei normali commenti. Una parte importante del processo di utilizzo delle annotazioni è costituita dalla creazione e dall'utilizzo degli *elaboratori di annotazioni*. Java SE5 mette a disposizione estensioni alle API di riflessione per la creazione di questi strumenti, nonché uno strumento esterno, chiamato **apt**, per l'analisi del codice sorgente Java con annotazioni.

Considerate il seguente elaboratore di annotazioni molto semplice, che legge la classe annotata **PasswordUtils** e utilizza la riflessione per cercare i tag **@UseCase**. A fronte di un elenco di valori **id**, riporta i casi d'utilizzo trovati e segnala quelli eventualmente mancanti.

```
//: annotations/UseCaseTracker.java
import java.lang.reflect.*;
import java.util.*;
```

```
public class UseCaseTracker {
  public static void
  trackUseCases(List<Integer> useCases, Class<?> cl) {
    for(Method m : cl.getDeclaredMethods()) {
      UseCase uc = m.getAnnotation(UseCase.class);
      if(uc != null) {
        System.out.println("Found Use Case: " + uc.id() +
          " " + uc.description());
        useCases.remove(new Integer(uc.id()));
      }
    }
    for(int i : useCases) {
      System.out.println("Warning: Missing use case - " + i);
    }
  }
  public static void main(String[] args) {
    List<Integer> useCases = new ArrayList<Integer>();
    Collections.addAll(useCases, 47, 48, 49, 50);
    trackUseCases(useCases, PasswordUtils.class);
  }
} /* Output:
Found Use Case:47 Passwords must contain at least one
numeric
Found Use Case:48 no description
Found Use Case:49 New passwords can't equal previously used
ones
Warning: Missing use case-50
*///:~
```

Questo codice utilizza sia il metodo **getDeclaredMethods()** di riflessione sia il metodo **getAnnotation()**, che deriva dall'interfaccia **AnnotatedElement**: anche classi come **Class**, **Method** e **Field** implementano la stessa interfaccia. Il metodo **getAnnotation()** restituisce l'oggetto di annotazione del tipo specificato, in questo caso "**UseCase**"; in assenza di annotazioni di quel tipo specifico, sul metodo annotato viene restituito **null**.

I valori dell'elemento vengono estratti chiamando **id()** e **description()**. Ricordate che a fronte del metodo **encryptPassword()** non è stata specificata

alcuna descrizione nell'annotazione, pertanto l'elaboratore rileva il valore predefinito **no description** quando chiama il metodo **description()** su quell'annotazione particolare.

## *Elementi dell'annotazione*

Il tag **@UseCase** definito in **UseCase.java** contiene l'elemento **int** chiamato **id** e l'elemento **String** chiamato **description**. I tipi possibili per gli elementi di annotazione sono elencati di seguito.

1. Tutti i tipi primitivi (**int**, **float**, **boolean** ecc.)
2. **String**
3. **Class**
4. **enum**
5. **Annotation**
6. Array contenenti un numero qualsiasi dei suddetti elementi

Se provate a utilizzare qualunque altro tipo il compilatore segnalerà un errore. Tenete presente che non è concesso ricorrere alle classi wrapper, tuttavia grazie alla funzionalità di autoboxing questa non è una vera limitazione. Potete anche avere elementi che sono essi stessi annotazioni: come vedrete in seguito, le annotazioni nidificate possono rivelarsi molto utili.

## *Limiti dei valori predefiniti*

Il compilatore è abbastanza esigente riguardo ai valori predefiniti degli elementi. Infatti, per nessun elemento è ammesso un valore non specificato: questo significa che gli elementi devono avere valori predefiniti oppure forniti dalla classe che utilizza l'annotazione.

Esiste tuttavia un'altra limitazione, per la quale nessun elemento di tipo non primitivo può accettare **null** come valore, sia esso dichiarato nel codice sorgente o definito come valore predefinito nell'interfaccia di annotazione. Questo limite complica la realizzazione di un elaboratore che agisce sulla base della presenza o assenza di un elemento, dal momento che ogni elemento è effettivamente presente in tutte le dichiarazioni di annotazione. Potete ovviare a questa limitazione controllando se sono presenti valori specifici, quali stringhe vuote o valori negativi.

```
//: annotations/SimulatingNull.java
import java.lang.annotation.*;
```

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SimulatingNull {
  public int id() default -1;
  public String description() default "";
} ///:~
```

Questa è una tipica forma idiomatica utilizzata nelle definizioni delle annotazioni.

## *Generare file esterni*

Le annotazioni sono particolarmente utili nell'ambito di strutture che esigono la presenza di informazioni supplementari a supporto del codice sorgente. Tecnologie quali Enterprise JavaBeans (prima di EJB3) richiedono numerose interfacce e descrittori che sono codice "riciclabile", definito nello stesso modo per ogni "bean".[3]

Allo stesso modo, i servizi web, le librerie di tag personalizzati e gli strumenti di mappatura relazionale/OOP come Toplink e Hibernate richiedono spesso descrittori XML esterni al codice. Dopo la definizione di una classe Java, il programmatore deve sottostare al supplizio di specificare nuovamente informazioni quali il nome, il package e così via, dati che già esistono nella classe originale. Ogni volta che utilizzate un file di descrittore esterno vi ritrovate con due fonti separate di informazioni su una classe, una situazione che può dare luogo a problemi di sincronizzazione del codice. Oltretutto, questo implica che i programmatori che lavorano al progetto debbano conoscere, oltre al linguaggio Java, anche le tecniche per modificare i descrittori.

Supponete di volere implementare una funzionalità di base per la mappatura relazionale/OOP, allo scopo di automatizzare la creazione di una tabella di database in cui memorizzare un JavaBean. Potreste utilizzare un file di descrittore XML per specificare il nome della classe e di ogni membro e le informazioni sulla mappatura al database; servendovi delle annotazioni, tuttavia, potete mantenere le informazioni nel file sorgente del JavaBean. A questo scopo vi occorrono le annotazioni per definire il nome della tabella di database connessa con il bean, le colonne e i tipi SQL che corrispondono alle proprietà del bean.

3. Componenti software contenenti una classe Java, utilizzati nello sviluppo di applicazioni web, che consentono l'incapsulamento di codice riutilizzabile.

Di seguito è mostrata un'annotazione per un bean che indica all'elaboratore di annotazioni di creare una tabella di database.

```
//: annotations/database/DBTable.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.TYPE) // Si applica soltanto alle classi
@Retention(RetentionPolicy.RUNTIME)
public @interface DBTable {
  public String name() default "";
} ///:~
```

Ogni **ElementType** specificato nell'annotazione **@Target** è una limitazione per segnalare al compilatore che l'annotazione può essere applicata soltanto a quel tipo particolare. Potete specificare un solo valore di **enum ElementType**, oppure un elenco separato da virgole di qualsiasi combinazione di valori. Se desiderate applicare l'annotazione a qualunque **ElementType** potrete omettere l'annotazione **@Target**, benché questa pratica sia poco comune.

Notate che **@DBTable** ha un elemento **name()**, in modo che l'annotazione possa fornire un nome per la tabella di database che verrà creata dall'elaboratore.

Di seguito trovate le annotazioni per i campi del JavaBean.

```
//: annotations/database/Constraints.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Constraints {
  boolean primaryKey() default false;
  boolean allowNull() default true;
  boolean unique() default false;
} ///:~
```

```
//: annotations/database/SQLString.java
package annotations.database;
import java.lang.annotation.*;
```

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLString {
  int value() default 0;
  String name() default "";
  Constraints constraints() default @Constraints;
} ///:~

//: annotations/database/SQLInteger.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLInteger {
  String name() default "";
  Constraints constraints() default @Constraints;
} ///:~
```

L'annotazione **@Constraints** consente all'elaboratore di estrarre i metadati sulla tabella di database. Questo è soltanto un piccolo sottoinsieme dei vincoli che caratterizzano i database, ma sufficiente per darvi un'indicazione del funzionamento.

Agli elementi **primaryKey()**, **allowNull()** e **unique()** vengono assegnati valori predefiniti adeguati, pertanto nella maggior parte dei casi l'utente dell'annotazione non dovrà digitare molto codice.

Le altre due annotazioni **@interface** definiscono i tipi SQL. Di nuovo, affinché questa struttura sia più utile dovete definire un'annotazione per ogni tipo SQL supplementare: in questo caso, due tipi saranno sufficienti.

Ognuno di questi tipi possiede un elemento **name**() e un elemento **constraints()**; quest'ultimo si serve della funzionalità di annotazione nidificata per includere informazioni sui vincoli del database per il tipo di colonna.

Notate che il valore predefinito per l'elemento **contraints()** è **@Constraints**. Poiché dopo questo tipo di annotazione non vi sono valori specificati tra parentesi per l'elemento, il valore predefinito **constraints()** è effettivamente un'annotazione **@Constraints** con il suo insieme di valori predefiniti. Per fare in modo che un'annotazione **@Constraints** nidificata con il suo insieme

di valori univoci venga impostata a **true** in modo predefinito, potete definire il relativo elemento come indicato di seguito.

```
//: annotations/database/Uniqueness.java
// Esempio di annotazioni nidificate
package annotations.database;

public @interface Uniqueness {
  Constraints constraints()
    default @Constraints(unique=true);
} ///:~
```

Il semplice bean mostrato di seguito si serve di queste annotazioni.

```
//: annotations/database/Member.java
package annotations.database;

@DBTable(name = "MEMBER")
public class Member {
  @SQLString(30) String firstName;
  @SQLString(50) String lastName;
  @SQLInteger Integer age;
  @SQLString(value = 30,
  constraints = @Constraints(primaryKey = true))
  String handle;
  static int memberCount;
  public String getHandle() { return handle; }
  public String getFirstName() { return firstName; }
  public String getLastName() { return lastName; }
  public String toString() { return handle; }
  public Integer getAge() { return age; }
} ///:~
```

All'annotazione di classe **@DBTable** è assegnato il valore **"MEMBER"**, che sarà utilizzato come nome della tabella. Le proprietà del bean, **firstName** e **lastName**, sono entrambe annotate con **@SQLString** e hanno valori di elemento pari, rispettivamente, a 30 e a 50. Queste annotazioni sono interessanti per due motivi: innanzitutto utilizzano il valore predefinito sull'annotazio-

ne **@Constraints** nidificata e, in secondo luogo, ricorrono a una funzionalità abbreviata.

Se definite un elemento su un'annotazione con il nome **value**, finché rimane l'unico tipo di elemento specificato non dovrete applicare necessariamente la sintassi della coppia nome-valore, ma potrete specificare soltanto il valore tra parentesi. Tale regola è applicabile a uno qualsiasi dei tipi di elemento ammessi. Naturalmente questo costituisce un limite nella chiamata al vostro elemento "value", ma nel caso descritto consente di specificare annotazioni facili da leggere e significative dal punto di vista semantico.

```
@SQLString(30)
```

L'elaboratore utilizzerà questo valore per impostare le dimensioni della colonna SQL che verrà creata.

Per quanto accurata sia la sintassi dei valori predefiniti, diventa rapidamente molto complessa. Considerate il campo **handle**: ha un'annotazione **@SQLString**, ma deve anche essere una chiave primaria del database, pertanto è necessario che il tipo di elemento **primaryKey** sia impostato all'annotazione nidificata **@Constraint**. È in situazioni come questa che la sintassi diventa molto confusa.

A questo punto, per questa annotazione nidificata siete costretti a utilizzare la forma estesa che prevede la coppia nome-valore, specificando di nuovo il nome dell'elemento e quello di **@interface**. Tuttavia, dal momento che l'elemento **value** è denominato in modo speciale, esso non è più l'unico valore di elemento specificato, di conseguenza non potete servirvi della forma abbreviata. Come vedete, il risultato è tutt'altro che elegante.

## Soluzioni alternative

Esistono altri modi per creare annotazioni per questa operazione. Per esempio, potreste avere un'unica classe di annotazione chiamata **@TableColumn** con un elemento **enum** che definisca valori quali **STRING**, **INTEGER**, **FLOAT** ecc. Questa soluzione elimina l'esigenza di avere un'**@interface** per ogni tipo SQL, ma rende impossibile qualificare i tipi con elementi aggiuntivi quali le dimensioni (*size*) o la precisione (*precision*), come sarebbe probabilmente più utile.

Potreste anche utilizzare un elemento **String** per descrivere il tipo SQL effettivo, per esempio "VARCHAR (30)" o "INTEGER". Così facendo potreste qualificare i tipi ma sareste vincolati alla corrispondenza, a livello di codice, tra tipi Java e tipi SQL, una scelta progettuale poco appropriata. È preferibi-

le evitare di ricompilare le vostre classi se cambiate database; sarebbe pratico poter indicare all'elaboratore di annotazioni che state utilizzando una variante di SQL diversa, e lasciare che il sistema si occupi dei dettagli quando elabora le annotazioni.

Una terza soluzione consiste nell'utilizzo combinato di entrambi i tipi di annotazione, **@Constraints** e il relativo tipo SQL, per esempio **@SQLInteger**, per annotare un determinato campo. Questa opzione è abbastanza confusa, anche se il compilatore consente un numero illimitato di annotazioni su un'annotazione di destinazione. Ricordate che utilizzando annotazioni multiple non potete servirvi due volte della stessa annotazione.

### *Le annotazioni non supportano l'ereditarietà*

Non è concesso utilizzare la parola chiave **extends** con **@interface**. È un vero peccato, poiché sarebbe stato utile poter definire un'annotazione **@TableColumn**, come suggerito, con un'annotazione nidificata di tipo **@SQLType**: in questo modo avreste potuto ereditare tutti i vostri tipi SQL, quali **@SQLInteger** e **@SQLString**, direttamente da **@SQLType**. Questa soluzione ridurrebbe la quantità di codice necessaria e renderebbe più ordinata la sintassi. Non vi è alcuna indicazione che nelle future versioni di Java le annotazioni potranno supportare l'ereditarietà, pertanto gli esempi suggeriti sembrano essere le tecniche migliori attualmente disponibili.

### *Implementazione dell'elaboratore*

Di seguito è presentato un esempio di elaboratore di annotazioni, che legge un file di classe, ne controlla le annotazioni di database e genera i comandi SQL per la creazione del database stesso.

```
//: annotations/database/TableCreator.java
// Elaboratore di associazioni basato sulla riflessione.
// {Args: annotations.database.Member}
package annotations.database;
import java.lang.annotation.*;
import java.lang.reflect.*;
import java.util.*;

public class TableCreator {
  public static void main(String[] args) throws Exception {
    if(args.length < 1) {
```

```
    System.out.println("arguments: annotated classes");
    System.exit(0);
  }
  for(String className : args) {
    Class<?> cl = Class.forName(className);
    DBTable dbTable = cl.getAnnotation(DBTable.class);
    if(dbTable == null) {
      System.out.println(
        "No DBTable annotations in class " + className);
      continue;
    }
    String tableName = dbTable.name();
    // Se il nome e' vuoto usa quello di Class:
    if(tableName.length() < 1)
      tableName = cl.getName().toUpperCase();
    List<String> columnDefs = new ArrayList<String>();
    for(Field field : cl.getDeclaredFields()) {
      String columnName = null;
      Annotation[] anns = field.getDeclaredAnnotations();
      if(anns.length < 1)
        continue; // Non e' una colonna della tabella del
                  // database
      if(anns[0] instanceof SQLInteger) {
        SQLInteger sInt = (SQLInteger) anns[0];
        // Se il nome non e' specificato, usa quello del campo
        if(sInt.name().length() < 1)
          columnName = field.getName().toUpperCase();
        else
          columnName = sInt.name();
        columnDefs.add(columnName + " INT" +
          getConstraints(sInt.constraints()));
      }
      if(anns[0] instanceof SQLString) {
        SQLString sString = (SQLString) anns[0];
        // Se il nome non e' specificato, usa quello
        // del campo.
```

```
          if(sString.name().length() < 1)
            columnName = field.getName().toUpperCase();
          else
            columnName = sString.name();
          columnDefs.add(columnName + " VARCHAR(" +
            sString.value() + ")" +
            getConstraints(sString.constraints()));
        }
        StringBuilder createCommand = new StringBuilder(
          "CREATE TABLE " + tableName + "(");
        for(String columnDef : columnDefs)
          createCommand.append("\n    " + columnDef + ",");
        // Gestisce la fine dell'istruzione SQL
        String tableCreate = createCommand.substring(
          0, createCommand.length() - 1) + ");";
        System.out.println("Table Creation SQL for " +
          className + " is:\n" + tableCreate);
      }
    }
  }
  private static String getConstraints(Constraints con) {
    String constraints = "";
    if(!con.allowNull())
      constraints += " NOT NULL";
    if(con.primaryKey())
      constraints += " PRIMARY KEY";
    if(con.unique())
      constraints += " UNIQUE";
    return constraints;
  }
} /* Output:
Table Creation SQL for annotations.database.Member is:
CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30));
Table Creation SQL for annotations.database.Member is:
CREATE TABLE MEMBER(
```

```
    FIRSTNAME VARCHAR(30),
    LASTNAME VARCHAR(50));
Table Creation SQL for annotations.database.Member is:
CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30),
    LASTNAME VARCHAR(50),
    AGE INT);
Table Creation SQL for annotations.database.Member is:
CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30),
    LASTNAME VARCHAR(50),
    AGE INT,
    HANDLE VARCHAR(30) PRIMARY KEY);
*///:~
```

Il metodo **main()** itera i nomi di classe forniti dalla riga di comando. Ogni classe viene caricata tramite **forName()** e controllata, per verificare la presenza dell'annotazione **@DBTable** con **getAnnotation(DBTable.class)**. Il nome della tabella viene trovato e memorizzato, poi tutti i campi nella classe vengono caricati e controllati tramite **getDeclaredAnnotations()**.

Questo metodo restituisce un array di tutte le annotazioni definite per un determinato metodo. L'operatore **instanceof** determina se tali annotazioni sono di tipo **@SQLInteger** e **@SQLString**, nel qual caso viene poi creato il corrispondente frammento **String** con il nome della colonna di tabella.

Ricordate che, non disponendo dell'ereditarietà per le interfacce di annotazione, l'utilizzo del metodo **getDeclaredAnnotations()** è il solo modo per emulare un comportamento polimorfico.

L'annotazione **@Constraint** nidificata viene poi passata a **getConstraints()**, che assembla una **String** contenente i *vincoli* (*constraint*) SQL.

È opportuno segnalare il fatto che questa tecnica è un modo in un certo senso "ingenuo" per definire una mappatura relazionale/OOP. Dal momento che l'annotazione di tipo **@DBTable** accetta il nome della tabella come parametro, se volete cambiare questo nome sarete costretti a ricompilare il codice Java, e questo potrebbe rivelarsi non opportuno. Esistono diverse strutture già predisposte per la mappatura degli oggetti ai database relazionali, molte delle quali si servono delle annotazioni.

**Esercizio 1** (2) Implementate altri tipi SQL nell'esempio del database.

**Progetto** Modificate l'esempio del database in modo che si colleghi e interagisca con un vero database, utilizzando il driver JDBC.

**Progetto** Modificate l'esempio del database in modo che crei file XML adeguati, invece del codice SQL.[4]

## Utilizzo di apt per l'elaborazione delle annotazioni

L'*elaboratore di annotazioni* **apt** è la prima versione Sun di un software di supporto all'elaborazione delle annotazioni. Trattandosi di una *première* lo strumento è ancora primitivo, pur presentando caratteristiche potenzialmente molto utili.

Come **javac**, **apt** è stato concepito per operare sui file sorgente Java invece che sulle classi compilate. In modalità predefinita **apt** compila i file sorgente al termine dell'elaborazione; questa caratteristica è pratica se state creando automaticamente nuovi sorgenti come parte del processo di compilazione: infatti, **apt** controlla le annotazioni nei file sorgente appena creati e li compila, il tutto simultaneamente.

Quando il vostro elaboratore di annotazioni crea un nuovo sorgente, il file viene verificato per controllare se contiene annotazioni, in un nuovo *round* (questo è il termine utilizzato nella documentazione) di elaborazione; **apt** continuerà un round dopo l'altro fino all'esaurimento di tutti i file sorgente creati e soltanto alla fine li compilerà tutti.

Ogni annotazione che scrivete necessita di un suo elaboratore, ma questo non è un problema poiché **apt** può raggruppare diversi elaboratori di annotazioni e consente di specificare in un'unica soluzione tutte le classi da elaborare: un approccio certamente più semplice rispetto all'iterazione manuale di tutte le classi **File** coinvolte. Potete anche implementare alcuni *listener* che vi informeranno del completamento dei round di elaborazione di un'annotazione.

Al momento di andare in stampa, **apt** non è disponibile come operazione Ant, in ogni caso può essere eseguito come operazione esterna: per maggiori dettagli consultate il supplemento all'indirizzo *http://mindview.net/Books/BetterJava*. Per compilare gli elaboratori di annotazione utilizzati in questo

4. I progetti sono proposte da utilizzare, per esempio, come pianificazioni a termine. Le soluzioni ai progetti non sono incluse nella guida delle soluzioni agli esercizi.

paragrafo il vostro CLASSPATH dovrà includere **tools.jar**, una libreria che contiene anche le interfacce **com.sun.mirror.***.

Lo strumento **apt** utilizza l'interfaccia **AnnotationProcessorFactory** per creare il tipo di elaboratore adatto per ogni annotazione trovata. Quando eseguite **apt** dovete specificare una classe factory oppure un CLASSPATH in cui il programma possa trovare le factory necessarie; in caso contrario **apt** inizierà un oscuro processo di "scoperta" (*discovery*), dettagliatamente descritto nella sezione *Developing an Annotation Processor* della documentazione Sun (*http://java.sun.com/j2se/1.5.0/docs/guide/apt/GettingStarted.html*).

Quando create un elaboratore di annotazioni da utilizzare con **apt** non potete utilizzare le funzionalità di riflessione di Java, poiché state lavorando con il codice sorgente, non con le classi compilate.[5]

Le API **mirror** risolvono questo problema, consentendovi di visualizzare i metodi, i campi e i tipi presenti nel codice sorgente non compilato.

Questa annotazione può essere utilizzata per estrarre i metodi **public** da una classe, e poi trasformarli in un'interfaccia.

```
//: annotations/ExtractInterface.java
// Elaborazione di annotazioni basata su APT.
package annotations;
import java.lang.annotation.*;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface ExtractInterface {
  public String value();
} ///:~
```

**RetentionPolicy** è impostato a **SOURCE**, considerato che non vi è ragione per conservare questa annotazione nel file di classe dopo averne estratta l'interfaccia.

La classe seguente fornisce un metodo **public** che può diventare parte di un'utile interfaccia.

5. Tuttavia, l'opzione non standard **-XclassesAsDecls** consente di elaborare anche le annotazioni che si trovano nelle classi compilate.

```
//: annotations/Multiplier.java
// Elaborazione di annotazioni basata su APT.
package annotations;

@ExtractInterface("IMultiplier")
public class Multiplier {
  public int multiply(int x, int y) {
    int total = 0;
    for(int i = 0; i < x; i++)
      total = add(total, y);
    return total;
  }
  private int add(int x, int y) { return x + y; }
  public static void main(String[] args) {
    Multiplier m = new Multiplier();
    System.out.println("11*16 = " + m.multiply(11, 16));
  }
} /* Output:
11*16 = 176
*///:~
```

La classe **Multiplier**, che funziona soltanto con numeri interi positivi, ha un metodo **multiply()** che chiama il metodo **private add()** diverse volte per eseguire la moltiplicazione; non essendo **public**, il metodo **add()** non fa parte dell'interfaccia. All'annotazione è assegnato il valore **IMultiplier**, che è il nome dell'interfaccia da creare.

A questo punto vi occorre un elaboratore che esegua l'estrazione.

```
//: annotations/InterfaceExtractorProcessor.java
// Elaborazione di annotazioni basata su APT.
// {Exec: apt -factory
// annotations.InterfaceExtractorProcessorFactory
// Multiplier.java -s ../annotations}
package annotations;
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import java.io.*;
```

```
import java.util.*;

public class InterfaceExtractorProcessor
  implements AnnotationProcessor {
  private final AnnotationProcessorEnvironment env;
  private ArrayList<MethodDeclaration> interfaceMethods =
    new ArrayList<MethodDeclaration>();
  public InterfaceExtractorProcessor(
    AnnotationProcessorEnvironment env) { this.env = env; }
  public void process() {
    for(TypeDeclaration typeDecl :
      env.getSpecifiedTypeDeclarations()) {
      ExtractInterface annot =
        typeDecl.getAnnotation(ExtractInterface.class);
      if(annot == null)
        break;
      for(MethodDeclaration m : typeDecl.getMethods())
        if(m.getModifiers().contains(Modifier.PUBLIC) &&
           !(m.getModifiers().contains(Modifier.STATIC)))
          interfaceMethods.add(m);
      if(interfaceMethods.size() > 0) {
        try {
          PrintWriter writer =
            env.getFiler().createSourceFile(annot.value());
          writer.println("package " +
            typeDecl.getPackage().getQualifiedName() +";");
          writer.println("public interface " +
            annot.value() + " {");
          for(MethodDeclaration m : interfaceMethods) {
            writer.print("  public ");
            writer.print(m.getReturnType() + " ");
            writer.print(m.getSimpleName() + " (");
            int i = 0;
            for(ParameterDeclaration parm :
              m.getParameters()) {
              writer.print(parm.getType() + " " +
                parm.getSimpleName());
```

```
                if(++i < m.getParameters().size())
                  writer.print(", ");
              }
              writer.println(");");
            }
            writer.println("}");
            writer.close();
          } catch(IOException ioe) {
            throw new RuntimeException(ioe);
          }
        }
      }
    }
  } ///:~
```

Il metodo **process()** è il punto in cui si svolge il lavoro vero e proprio. La classe **MethodDeclaration** e il metodo **getModifiers()** servono per individuare i metodi **public** della classe elaborata, ignorando quelli **static**. I metodi **public** trovati vengono memorizzati in un **ArrayList** e utilizzati per creare i metodi di una nuova definizione d'interfaccia in un file **.java**.

Notate che al costruttore viene passato un oggetto **AnnotationProcessorEnvironment**, che può essere interrogato per conoscere tutti i tipi (definizioni della classe) in corso di elaborazione da parte di **apt**, in modo da ottenere gli oggetti **Messager** e **Filer**. L'oggetto **Messager** consente di segnalare messaggi all'utente, per esempio gli eventuali errori che potrebbero verificarsi in fase di elaborazione e la loro posizione nel codice sorgente. **Filer** è una sorta di **PrintWriter** mediante il quale potrete creare nuovi file: il motivo principale per utilizzare questo oggetto in luogo di **PrintWriter** è che **Filer** consente ad **apt** di tenere traccia di tutti i nuovi file creati, in modo da poterli controllare alla ricerca di annotazioni e compilarli, se necessario.

Osservate anche che il metodo **createSourceFile()** apre un normale flusso di output con il nome appropriato per la vostra classe o interfaccia Java. Non essendo disponibile il supporto per la creazione di costrutti Java, dovrete generare il codice sorgente servendovi dei metodi **print()** e **println()**, in un certo senso primordiali: questo significa che dovrete assicurarvi che non vi siano errori di parentesi e che la sintassi sia valida.

Il metodo **process()** è chiamato dallo strumento **apt**, che necessita di una factory per fornire l'elaboratore corretto.

```
//: annotations/InterfaceExtractorProcessorFactory.java
// Elaborazione di annotazioni basata su APT.
package annotations;
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import java.util.*;

public class InterfaceExtractorProcessorFactory
  implements AnnotationProcessorFactory {
  public AnnotationProcessor getProcessorFor(
    Set<AnnotationTypeDeclaration> atds,
    AnnotationProcessorEnvironment env) {
    return new InterfaceExtractorProcessor(env);
  }
  public Collection<String> supportedAnnotationTypes() {
    return
     Collections.singleton("annotations.ExtractInterface");
  }
  public Collection<String> supportedOptions() {
    return Collections.emptySet();
  }
} ///:~
```

L'interfaccia **AnnotationProcessorFactory** dispone soltanto di tre metodi. Come vedete, quello fornito dall'elaboratore è **getProcessorFor()**: il metodo accetta un **Set** di dichiarazioni di tipo (classi Java a fronte delle quali viene utilizzato **apt**) e l'oggetto **AnnotationProcessorEnvironment**, che già avete visto trattare dall'elaboratore. Gli altri due metodi, **supportedAnnotationTypes()** e **supportedOptions()**, consentono di controllare che per ogni annotazione trovata da **apt** sia disponibile un elaboratore e che tutte le opzioni specificate nel prompt di comando siano supportate.

Il metodo **getProcessorFor()** è particolarmente importante: infatti, se non restituite il nome completo della classe del vostro tipo di annotazione nella collezione, **String**, **apt** vi informerà che non è disponibile il relativo elaboratore e terminerà senza eseguire alcunché.

L'elaboratore e la factory sono nel pacchetto **annotations**, pertanto, per la struttura di directory indicata, l'intera riga di comando è inclusa nel tag di commento "**Exec**" all'inizio di **InterfaceExtractorProcessor.java**. Questo

consente ad **apt** di utilizzare la classe factory definita in precedenza (**InterfaceExtractorProcessorFactory**) e di elaborare il file **Multiplier.java**. L'opzione **-s** indica di creare ogni nuovo file nella directory **annotations**. Come potete dedurre osservando le istruzioni **println()** nell'elaboratore **InterfaceExtractorProcessor**, il file **IMultiplier.java** generato sarà simile al seguente:

```
package annotations;
public interface IMultiplier {
 public int multiply (int x, int y);
}
```

Questo file sarà anche compilato da **apt**, pertanto nella stessa directory troverete il file **IMultiplier.class**.

> **Esercizio 2** (3) Aggiungete il supporto per la divisione all'estrattore d'interfaccia.

## Utilizzo del modello Visitor con apt

L'elaborazione delle annotazioni può diventare complessa. L'esempio precedente si riferiva a un elaboratore relativamente semplice che interpreta una sola annotazione, il cui funzionamento, tuttavia, è piuttosto complesso. Per impedire che tale complessità aumenti a dismisura con il crescere del numero di annotazioni e di elaboratori, le API **mirror** forniscono le classi di supporto al design pattern *Visitor*. Questo è uno dei modelli classici descritti in *Design Patterns*, di Gamma, Helm, Johnson e Vlissides (Addison-Wesley, 1995), di cui troverete una spiegazione più dettagliata in *Thinking in Patterns.*

Un modello Visitor percorre una struttura di dati o una collezione di oggetti, eseguendo un'operazione su ogni elemento trovato. La struttura dati non deve essere necessariamente ordinata, e l'operazione eseguita su ogni oggetto è specifica del suo tipo. Questo approccio consente di scorporare le operazioni dagli oggetti stessi, consentendovi, di fatto, di implementare nuove operazioni senza aggiungere nuovi metodi alle definizioni di classe.

Questo rende il pattern Visitor particolarmente pratico per elaborare le annotazioni, dal momento che una classe Java può essere considerata come una raccolta di oggetti quali **TypeDeclaration**, **FieldDeclaration**, **MethodDeclaration** e così via. Quando utilizzate lo strumento **apt** con il modello Visitor, fornite una classe **Visitor** che ha un metodo per gestire ogni tipo di dichiarazione incontrata; in questo modo potete implementare il comportamento adatto per le annotazioni su metodi, classi, campi ecc.

Di seguito è mostrata una nuova versione del generatore di tabelle SQL, che integra una factory e un elaboratore che utilizza il modello Visitor.

```
//: annotations/database/TableCreationProcessorFactory.java
// L'esempio del database, rielaborato con l'utilizzo del
// pattern Visitor.
// {Exec: apt -factory
// annotations.database.TableCreationProcessorFactory
// database/Member.java -s database}
package annotations.database;
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import com.sun.mirror.util.*;
import java.util.*;
import static com.sun.mirror.util.DeclarationVisitors.*;

public class TableCreationProcessorFactory
  implements AnnotationProcessorFactory {
  public AnnotationProcessor getProcessorFor(
    Set<AnnotationTypeDeclaration> atds,
    AnnotationProcessorEnvironment env) {
    return new TableCreationProcessor(env);
  }
  public Collection<String> supportedAnnotationTypes() {
    return Arrays.asList(
      "annotations.database.DBTable",
      "annotations.database.Constraints",
      "annotations.database.SQLString",
      "annotations.database.SQLInteger");
  }
  public Collection<String> supportedOptions() {
    return Collections.emptySet();
  }
  private static class TableCreationProcessor
    implements AnnotationProcessor {
    private final AnnotationProcessorEnvironment env;
    private String sql = "";
```

```
public TableCreationProcessor(
  AnnotationProcessorEnvironment env) {
  this.env = env;
}
public void process() {
  for(TypeDeclaration typeDecl :
    env.getSpecifiedTypeDeclarations()) {
    typeDecl.accept(getDeclarationScanner(
      new TableCreationVisitor(), NO_OP));
    sql = sql.substring(0, sql.length() - 1) + ");";
    System.out.println("creation SQL is: \n" + sql);
    sql = "";
  }
}
private class TableCreationVisitor
  extends SimpleDeclarationVisitor {
  public void visitClassDeclaration(
    ClassDeclaration d) {
    DBTable dbTable = d.getAnnotation(DBTable.class);
    if(dbTable != null) {
      sql += "CREATE TABLE ";
      sql += (dbTable.name().length() < 1)
        ? d.getSimpleName().toUpperCase()
        : dbTable.name();
      sql += " (";
    }
  }
  public void visitFieldDeclaration(
    FieldDeclaration d) {
    String columnName = "";
    if(d.getAnnotation(SQLInteger.class) != null) {
      SQLInteger sInt = d.getAnnotation(
          SQLInteger.class);
      // Se il nome non e' specificato, utilizza quello
      // del campo.
      if(sInt.name().length() < 1)
```

```
            columnName = d.getSimpleName().toUpperCase();
          else
            columnName = sInt.name();
          sql += "\n    " + columnName + " INT" +
            getConstraints(sInt.constraints()) + ",";
        }
        if(d.getAnnotation(SQLString.class) != null) {
          SQLString sString = d.getAnnotation(
              SQLString.class);
          // Se il nome non e' specificato, utilizza quello
          // del campo.
          if(sString.name().length() < 1)
            columnName = d.getSimpleName().toUpperCase();
          else
            columnName = sString.name();
          sql += "\n    " + columnName + " VARCHAR(" +
            sString.value() + ")" +
            getConstraints(sString.constraints()) + ",";
        }
      }
      private String getConstraints(Constraints con) {
        String constraints = "";
        if(!con.allowNull())
          constraints += " NOT NULL";
        if(con.primaryKey())
          constraints += " PRIMARY KEY";
        if(con.unique())
          constraints += " UNIQUE";
        return constraints;
      }
    }
  }
} ///:~
```

Una volta eseguito il programma, il risultato sarà identico all'esempio precedente che si serve di **DBTable**.

In questo esempio l'elaboratore e Visitor sono classi interne; notate che il metodo **process()** si limita ad aggiungere la classe Visitor e inizializzare la stringa SQL.

Entrambi i parametri di **getDeclarationScanner()** sono Visitor: il primo è utilizzato prima dell'inizializzazione di ogni dichiarazione, il secondo in seguito. Questo elaboratore richiede soltanto il Visitor "pre-analisi", pertanto come secondo parametro è indicato **NO_OP**; questo parametro è un campo **static** nell'interfaccia **DeclarationVisitor**, che è un **DeclarationVisitor** che non esegue alcunché.

La classe **TableCreationVisitor** estende **SimpleDeclarationVisitor**, sovrascrivendo i due metodi **visitClassDeclaration()** e **visitFieldDeclaration()**; la classe **SimpleDeclarationVisitor** è un adattatore che implementa tutti i metodi dell'interfaccia **DeclarationVisitor**, affinché possiate scegliere quelle che vi occorrono.

Nel metodo **visitClassDeclaration()** l'oggetto **ClassDeclaration** viene controllato per verificare la presenza dell'annotazione **DBTable**: nel caso essa esista, verrà inizializzata la prima parte della **String** di creazione delle istruzioni SQL. In **visitFieldDeclaration()** viene interrogata la dichiarazione di campo alla ricerca delle relative annotazioni di campo: le informazioni vengono estratte in modo molto simile all'esempio originale.

Questa tecnica potrà sembrarvi più complessa, tuttavia produce una soluzione più scalabile. Se la complessità del vostro elaboratore di annotazioni aumenta, infatti, la realizzazione di un elaboratore autonomo come quello dell'esempio precedente potrebbe diventare straordinariamente complicata.

**Esercizio 3** (2) Aggiungete il supporto a più tipi SQL in **TableCreationProcessorFactory.java**.

## Test unitari basati sulle annotazioni

I cosiddetti *test unitari* (*unit testing*) sono una serie di uno o più test creati per ogni metodo di una determinata classe, destinati a verificare regolarmente il comportamento corretto dei componenti della classe. Lo strumento Java più noto utilizzato per test unitari è chiamato *JUnit*.[6]

Uno dei problemi principali con le versioni di JUnit precedenti JUnit4 è il processo complicato richiesto per impostare ed eseguire i test. Questo incon-

6. In origine l'autore aveva pensato di creare un "JUnit migliorato", basato sui concetti di progettazione presentati in questo capitolo. Tuttavia sembra che JUnit4 includa molti di questi concetti, per cui vi sarà più semplice servirsi di questo prodotto.

veniente si è ridimensionato con il tempo, in ogni caso l'introduzione delle annotazioni rende le operazioni di test più consone all'idea del "più semplice sistema di test unitari in grado di funzionare".

Con le versioni di JUnit pre-annotazioni è necessario creare una classe separata per contenere i test unitari.

L'autore ha tuttavia scelto un approccio diverso, basato sulle annotazioni. Grazie alle annotazioni, infatti, potete includere i test unitari all'interno della classe da esaminare, riducendo così al minimo la durata e la difficoltà del test unitario. Questo meccanismo ha il vantaggio di consentire una facile verifica sia dei metodi **private** sia di quelli **public**.

Dal momento che questa struttura di test d'esempio è basata sulle annotazioni, è stata chiamata **@Unit**. La forma più elementare di test, che probabilmente utilizzerete più spesso, richiede soltanto di indicare con **@Test** i metodi da esaminare. Una possibilità consiste nel fare in modo che i metodi di test non accettino argomenti e restituiscano un valore **boolean** per indicare il successo o il fallimento del test. A questi metodi potete assegnare il nome che preferite; inoltre, i metodi di test **@Unit** possono avere qualsiasi tipo di accesso, compreso **private**.

Per utilizzare **@Unit** tutto ciò che dovete fare è importare il package **net.mindview.atunit**, contrassegnare i metodi e i campi appropriati con il tag di test **@Unit**, che vedrete meglio nei prossimi esempi, e mandare in esecuzione **@Unit** passando come argomento la classe risultante.[7]

Di seguito è mostrato un semplice esempio.

```
//: annotations/AtUnitExample1.java
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExample1 {
  public String methodOne() {
    return "This is methodOne";
  }
  public int methodTwo() {
    System.out.println("This is methodTwo");
```

7. La libreria **net.mindview.atunit** è inclusa nel codice a corredo di questo libro, scaricabile all'indirizzo *www.mindview.net*.

```
    return 2;
  }
  @Test boolean methodOneTest() {
    return methodOne().equals("This is methodOne");
  }
  @Test boolean m2() { return methodTwo() == 2; }
  @Test private boolean m3() { return true; }
  // Mostra l'output per gli errori:
  @Test boolean failureTest() { return false; }
  @Test boolean anotherDisappointment() { return false; }
  public static void main(String[] args) throws Exception {
    OSExecute.command(
      "java net.mindview.atunit.AtUnit AtUnitExample1");
  }
} /* Output:
annotations.AtUnitExample1
  . methodOneTest
  . m2 This is methodTwo

  . m3
  . failureTest (failed)
  . anotherDisappointment (failed)
(5 tests)

>>> 2 FAILURES<<<
  annotations.AtUnitExample1: failureTest
  annotations.AtUnitExample1: anotherDisappointment
*///:~
```

Le classi da sottoporre a test **@Unit** devono essere pacchettizzate.

L'annotazione **@Test** che precede i metodi **methodOneTest()**, **m2()**, **m3()**, **failureTest()** e **anotherDisappointment()** indica a **@Unit** di eseguire questi metodi come test unitari. Inoltre si assicura che tali metodi non accettino argomenti e restituiscano un valore **boolean** o **void**. Quando eseguite i testi unitari la vostra unica responsabilità è determinare se la prova riesce oppure no e, per i metodi che restituiscono **boolean**, se restituisce **true** o **false**.

Se avete familiarità con JUnit noterete anche che l'output di **@Unit** offre maggiori informazioni: elenca il test in esecuzione in quel momento e, al termine, segnala le classi e i test che non hanno superato le prove.

Non siete obbligati a includere i metodi di test all'interno delle vostre classi, se ritenete che questo non sia l'approccio appropriato. Il modo più semplice per creare test "non embedded" risiede nell'utilizzo dell'ereditarietà.

```
//: annotations/AtUnitExternalTest.java
// Creazione di test non embedded.
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExternalTest extends AtUnitExample1 {
  @Test boolean _methodOne() {
    return methodOne().equals("This is methodOne");
  }
  @Test boolean _methodTwo() { return methodTwo() == 2; }
  public static void main(String[] args) throws Exception {
    OSExecute.command(
     "java net.mindview.atunit.AtUnit AtUnitExternalTest");
  }
} /* Output:
annotations.AtUnitExternalTest
  . _methodOne
  . _methodTwo This is methodTwo

OK (2 testS)
*///:~
```

Questo esempio della libreria evidenzia anche l'importanza di una nomenclatura flessibile, in contrasto con JUnit, che richiede invece che i nomi di tutti i test inizino con la parola "**test**".

In questo caso, ai metodi **@Test** che stanno verificando direttamente un altro metodo viene assegnato il nome del metodo, prefissandolo con un carattere di sottolineatura (_): l'autore non intende suggerire che questa sia la tecnica ideale, tuttavia rappresenta una possibile soluzione.

Per creare test non embedded potete anche servirvi della composizione.

```
//: annotations/AtUnitComposition.java
// Creazione di test non embedded.
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitComposition {
  AtUnitExample1 testObject = new AtUnitExample1();
  @Test boolean _methodOne() {
    return
      testObject.methodOne().equals("This is methodOne");
  }
  @Test boolean _methodTwo() {
    return testObject.methodTwo() == 2;
  }
  public static void main(String[] args) throws Exception {
    OSExecute.command(
     "java net.mindview.atunit.AtUnit AtUnitComposition");
  }
} /* Output:
annotations.AtUnitComposition
  . _methodOne
  . _methodTwo This is methodTwo

OK (2 tests)
*///:~
```

Non esistono metodi "assert" speciali come in JUnit, ma la seconda forma del metodo **@Test** consente di restituire **void**, o **boolean** se volete restituire **true** o **false**. Per verificare se i test hanno avuto successo, potete utilizzare le istruzioni Java **assert**. In generale le asserzioni Java devono essere abilitate specificando l'opzione **-ea** sulla riga di comando di **java**; **@Unit**, invece, le abilita automaticamente. Per indicare la condizione di errore potete persino ricorrere a un'eccezione. Uno degli obiettivi progettuali di **@Unit** è stato quello di ridurre al minimo l'esigenza di sintassi aggiuntiva e gli **assert** e le eccezioni di Java sono tutto ciò che occorre per segnalare gli errori. Un'**as-**

**sert** fallita o un'eccezione risultante dal metodo di test vengono considerate come test falliti; tuttavia in questi casi **@Unit** non si ferma, proseguendo fino all'esaurimento dei test previsti, come nell'esempio seguente.

```
//: annotations/AtUnitExample2.java
// In @Tests si possono utilizzare asserzioni ed eccezioni.
package annotations;
import java.io.*;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExample2 {
  public String methodOne() {
    return "This is methodOne";
  }
  public int methodTwo() {
    System.out.println("This is methodTwo");
    return 2;
  }
  @Test void assertExample() {
    assert methodOne().equals("This is methodOne");
  }
  @Test void assertFailureExample() {
    assert 1 == 2: "What a surprise!";
  }
  @Test void exceptionExample() throws IOException {
    new FileInputStream("nofile.txt"); // Solleva un'eccezione
  }
  @Test boolean assertAndReturn() {
    // Asserzione con messaggio:
    assert methodTwo() == 2: "methodTwo must equal 2";
    return methodOne().equals("This is methodOne");
  }
  public static void main(String[] args) throws Exception {
    OSExecute.command(
      "java net.mindview.atunit.AtUnit AtUnitExample2");
  }
```

```
} /* Output:
annotations.AtUnitExample2
  . assertExample
  . assertFailureExample java.lang.AssertionError: What a
     surprise
(failed)
  . exceptionExample java.io.FileNotFoundException: nofile.txt
(No such file or directory)
(failed)
  . assertAndReturn This is methodTwo

(4 tests)

>>> 2 FAILURES <<<
  annotations.AtUnitExample2: assertFailureExample
  annotations.AtUnitExample2: exceptionExample
*///:~
```

Di seguito è mostrato un esempio che si serve di test non embedded con asserzioni, con cui esegue semplici verifiche di **java.util.HashSet**.

```
//: annotations/HashSetTest.java
package annotations;
import java.util.*;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class HashSetTest {
  HashSet<String> testObject = new HashSet<String>();
  @Test void initialization() {
    assert testObject.isEmpty();
  }
  @Test void _contains() {
    testObject.add("one");
    assert testObject.contains("one");
  }
  @Test void _remove() {
```

```
      testObject.add("one");
      testObject.remove("one");
      assert testObject.isEmpty();
    }
    public static void main(String[] args) throws Exception {
      OSExecute.command(
        "java net.mindview.atunit.AtUnit HashSetTest");
    }
  } /* Output:
  annotations.HashSetTest
    . initialization
    . _remove
    . _contains
  OK (3 tests)
  *///:~
```

In assenza di altri vincoli, il metodo che si avvale dell'ereditarietà sembrerebbe il più semplice.

**Esercizio 4** (3) Verificate che sia creato un nuovo **testObject** prima di ogni test.

**Esercizio 5** (1) Modificate l'esempio precedente al fine di utilizzare l'ereditarietà.

**Esercizio 6** (1) Eseguite test su **LinkedList** applicando la tecnica descritta in **HashSetTest.java**.

**Esercizio 7** (1) Modificate l'esercizio precedente al fine di utilizzare l'ereditarietà.

Per ogni test unitario, **@Unit** crea un oggetto della classe da verificare tramite il costruttore predefinito. Viene chiamato il test su questo oggetto, poi viene scartato per impedire che eventuali effetti secondari possano ripercuotersi su altri test unitari. Se non disponete di un costruttore predefinito, o se la creazione dei vostri progetti richiede un meccanismo di costruzione più elaborato, potrete utilizzare un metodo **static** per creare l'oggetto e collegarvi l'annotazione **@TestObjectCreate**, come nel seguente esempio.

```
//: annotations/AtUnitExample3.java
package annotations;
import net.mindview.atunit.*;
```

```
import net.mindview.util.*;

public class AtUnitExample3 {
  private int n;
  public AtUnitExample3(int n) { this.n = n; }
  public int getN() { return n; }
  public String methodOne() {
    return "This is methodOne";
  }
  public int methodTwo() {
    System.out.println("This is methodTwo");
    return 2;
  }
  @TestObjectCreate static AtUnitExample3 create() {
    return new AtUnitExample3(47);
  }
  @Test boolean initialization() { return n == 47; }
  @Test boolean methodOneTest() {
    return methodOne().equals("This is methodOne");
  }
  @Test boolean m2() { return methodTwo() == 2; }
  public static void main(String[] args) throws Exception {
    OSExecute.command(
      "java net.mindview.atunit.AtUnit AtUnitExample3");
  }
} /* Output:
annotations.AtUnitExample3
  . initialization
  . methodOneTest
  . m2 This is methodTwo

OK (3 tests)
*///:~
```

Il metodo **@TestObjectCreate** deve essere **static** e restituire un oggetto del tipo che state testando; **@Unit** si assicurerà che questi requisiti vengano rispettati.

A volte sono necessari campi supplementari per supportare il test unitario. L'annotazione **@TestProperty** può essere sfruttata per contrassegnare i campi utilizzati soltanto per il test unitario, in modo da poterli rimuovere prima di consegnare il prodotto al cliente. Ecco un esempio che legge valori da una **String** che viene suddivisa in varie parti, tramite il metodo **String.split()**; questi dati sono poi utilizzati per creare gli oggetti di test.

```
//: annotations/AtUnitExample4.java
package annotations;
import java.util.*;
import net.mindview.atunit.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class AtUnitExample4 {
  static String theory = "All brontosauruses " +
    "are thin at one end, much MUCH thicker in the " +
    "middle, and then thin again at the far end.";";
  private String word;
  private Random rand = new Random(); // "Seme" basato
                                      // sull'ora
  public AtUnitExample4(String word) { this.word = word; }
  public String getWord() { return word; }
  public String scrambleWord() {
    List<Character> chars = new ArrayList<Character>();
    for(Character c : word.toCharArray())
      chars.add(c);
    Collections.shuffle(chars, rand);
    StringBuilder result = new StringBuilder();
    for(char ch : chars)
      result.append(ch);
    return result.toString();
  }
  @TestProperty static List<String> input =
    Arrays.asList(theory.split(" "));
  @TestProperty
    static Iterator<String> words = input.iterator();
```

```
  @TestObjectCreate static AtUnitExample4 create() {
    if(words.hasNext())
      return new AtUnitExample4(words.next());
    else
      return null;
  }
  @Test boolean words() {
    print("'" + getWord() + "'");
    return getWord().equals("are");
  }
  @Test boolean scramble1() {
    // Si usa un "seme" specifico per ottenere risultati
    // verificabili:
    rand = new Random(47);
    print("'" + getWord() + "'");
    String scrambled = scrambleWord();
    print(scrambled);
    return scrambled.equals("lAl");
  }
  @Test boolean scramble2() {
    rand = new Random(74);
    print("'" + getWord() + "'");
    String scrambled = scrambleWord();
    print(scrambled);
    return scrambled.equals("tsaeborornussu");
  }
  public static void main(String[] args) throws Exception {
    System.out.println("starting");
    OSExecute.command(
      "java net.mindview.atunit.AtUnit AtUnitExample4");
  }
} /* Output:
Starting
annotations.AtUnitExample4
  . words 'All'
```

```
  . scramble1 'i'
IAI

  . scramble2 'brontosauruses'
tsaebororornussu

   . words 'are'

OK (3 tests)
*///:~
```

**@TestProperty** può essere sfruttato anche per contrassegnare metodi che pur essendo utilizzabili durante i test non sono test veri e propri.

Notate che questo programma si basa sull'ordine di esecuzione dei test, una pratica che di norma è sconsigliabile.

Se nella fase di creazione del vostro oggetto di test eseguite operazioni di inizializzazione che richiedono una successiva ripulitura, potrete eventualmente aggiungere un metodo **static @TestObjectCleanup**: esso eseguirà le operazioni di cleanup quando l'oggetto non servirà più. Nell'esempio seguente **@TestObjectCreate** apre un file per creare ogni oggetto di test; pertanto il file dovrà essere poi chiuso prima che l'oggetto di test venga scartato.

```
//: annotations/AtUnitExample5.java
package annotations;
import java.io.*;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExample5 {
  private String text;
  public AtUnitExample5(String text) { this.text = text; }
  public String toString() { return text; }
  @TestProperty static PrintWriter output;
  @TestProperty static int counter;
  @TestObjectCreate static AtUnitExample5 create() {
    String id = Integer.toString(counter++);
```

```
      try {
        output = new PrintWriter("Test" + id + ".txt");
      } catch(IOException e) {
        throw new RuntimeException(e);
      }
      return new AtUnitExample5(id);
    }
    @TestObjectCleanup static void
    cleanup(AtUnitExample5 tobj) {
      System.out.println("Running cleanup");
      output.close();
    }
    @Test boolean test1() {
      output.print("test1");
      return true;
    }
    @Test boolean test2() {
      output.print("test2");
      return true;
    }
    @Test boolean test3() {
      output.print("test3");
      return true;
    }
    public static void main(String[] args) throws Exception {
      OSExecute.command(
        "java net.mindview.atunit.AtUnit AtUnitExample5");
    }
  } /* Output:
  annotations.AtUnitExample5
    . test1
  Running cleanup
    . test2
  Running cleanup
    . test3
  Running cleanup
```

```
OK (3 tests)
*///:~
```

Come potete vedere dall'output, il metodo di cleanup viene eseguito automaticamente dopo ogni test.

## *Utilizzo di @Unit con i generici*

I generici rappresentano un problema particolare; non è possibile implementare "test generici" poiché devono essere eseguiti su un tipo di parametro o su un insieme di parametri ben definiti. La soluzione è semplice: ereditare la classe di test da una versione specificata della classe generica.

Di seguito è mostrata una semplice implementazione di uno stack.

```
//: annotations/StackL.java
// Uno stack costruito su una LinkedList.
package annotations;
import java.util.*;

public class StackL<T> {
  private LinkedList<T> list = new LinkedList<T>();
  public void push(T v) { list.addFirst(v); }
  public T top() { return list.getFirst(); }
  public T pop() { return list.removeFirst(); }
} ///:~
```

Per eseguire un test sulla versione **String** ereditate una classe di test da **StackL<String>**.

```
//: annotations/StackLStringTest.java
// Come applicare @Unit ai generici.
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class StackLStringTest extends StackL<String> {
  @Test void _push() {
    push("one");
```

```
    assert top().equals("one");
    push("two");
    assert top().equals("two");
  }
  @Test void _pop() {
    push("one");
    push("two");
    assert pop().equals("two");
    assert pop().equals("one");
  }
  @Test void _top() {
    push("A");
    push("B");
    assert top().equals("B");
    assert top().equals("B");
  }
  public static void main(String[] args) throws Exception {
    OSExecute.command(
      "java net.mindview.atunit.AtUnit StackLStringTest");
  }
} /* Output:
annotations.StackLStringTest
  . _push
  . _pop
  . _top
OK (3 tests)
*///:~
```

L'unico inconveniente potenziale dell'ereditarietà risiede nel fatto che viene persa la capacità di accedere ai metodi **private** presenti nella classe in fase di test. Se questo fosse un problema, potrete rendere **protected** il metodo in questione oppure aggiungere un metodo non **private @TestProperty** che chiama il metodo **private**.

Il metodo **@TestProperty** potrà essere eliminato dal codice finale per mezzo dello strumento **AtUnitRemover**, di cui si tratterà nel prosieguo del capitolo.

**Esercizio 8** (2) Create una classe con un metodo **private** e aggiungete un metodo non **private @TestProperty**, come descritto in precedenza. Chiamate questo metodo nel vostro codice di test.

**Esercizio 9** (2) Scrivete test **@Unit** di base per **HashMap**.

**Esercizio 10** (2) Selezionate un esempio qualsiasi del manuale e aggiungetevi test **@Unit**.

## Le suite non sono necessarie

Uno dei maggiori vantaggi che **@Unit** presenta rispetto a JUnit è che le suite sono inutili. In JUnit, infatti, dovete comunicare in qualche modo al software "che cosa" deve essere esaminato: questo richiede l'introduzione delle cosiddette *suite*, ovvero i raggruppamenti su cui si basa JUnit per localizzare i test da eseguire.

**@Unit**, di contro, si limita a cercare i file di classe che contengono le annotazioni appropriate, quindi esegue i metodi **@Test**. L'obiettivo principale dell'applicazione di test **@Unit** dell'autore è stato la massima trasparenza, in modo da potere iniziare a utilizzarla semplicemente aggiungendo metodi **@Test**, senza dovere integrare altro codice né disporre di competenze analoghe a quelle richieste da JUnit e da molti altri ambienti di test. È abbastanza difficile scrivere test senza aggiungere nuovi "paletti", e i test di **@Unit** cercano di semplificare al massimo questo compito: in questo modo è più probabile che lo utilizziate per scrivere le prove.

## Implementazione di @Unit

Per prima cosa dovete definire tutti i tipi di annotazione; si tratta di semplici tag che non hanno campi. Il tag **@Test** è stato definito all'inizio del capitolo; di seguito sono elencate le altre annotazioni.

```
//: net/mindview/atunit/TestObjectCreate.java
// Il tag @Unit @TestObjectCreate.
package net.mindview.atunit;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestObjectCreate {} ///:~
```

```
//: net/mindview/atunit/TestObjectCleanup.java
// Il tag @Unit @TestObjectCleanup.
package net.mindview.atunit;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestObjectCleanup {} ///:~
```

```
//: net/mindview/atunit/TestProperty.java
// Il tag @Unit @TestProperty.
package net.mindview.atunit;
import java.lang.annotation.*;

// Campi e metodi possono essere contrassegnati come
// proprieta':
@Target({ElementType.FIELD, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface TestProperty {} ///:~
```

Per tutti i test il valore di **Retention** è **RUNTIME** poiché il framework **@Unit** deve identificare i test nel codice compilato.

Per implementare il motore che esegue i test viene utilizzata la riflessione, che estrae le annotazioni; il programma si serve di queste informazioni per decidere come compilare gli oggetti di test ed eseguire questi ultimi. Grazie alle annotazioni il codice richiesto è sorprendentemente compatto e immediato.

```
//: net/mindview/atunit/AtUnit.java
// Un framework di test basato sulle annotazioni.
// {RunByHand}
package net.mindview.atunit;
import java.lang.reflect.*;
import java.io.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;
```

```
public class AtUnit implements ProcessFiles.Strategy {
  static Class<?> testClass;
  static List<String> failedTests= new ArrayList<String>();
  static long testsRun = 0;
  static long failures = 0;
  public static void main(String[] args) throws Exception {
    ClassLoader.getSystemClassLoader()
      .setDefaultAssertionStatus(true); // Abilita le
                                        // asserzioni
    new ProcessFiles(new AtUnit(), "class").start(args);
    if(failures == 0)
      print(" OK (" + testsRun + " tests)");
    else {
      print("(" + testsRun + " tests)");
      print("\n>>> " + failures + " FAILURE" +
        (failures > 1 ? "S" : "") + " <<<");
      for(String failed : failedTests)
        print("  " + failed);
    }
  }
  public void process(File cFile) {
    try {
      String cName = ClassNameFinder.thisClass(
        BinaryFile.read(cFile));
      if(!cName.contains("."))
        return; // Ignora le classi senza package
      testClass = Class.forName(cName);
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
    TestMethods testMethods = new TestMethods();
    Method creator = null;
    Method cleanup = null;
    for(Method m : testClass.getDeclaredMethods()) {
      testMethods.addIfTestMethod(m);
      if(creator == null)
```

```
    creator = checkForCreatorMethod(m);
  if(cleanup == null)
    cleanup = checkForCleanupMethod(m);
}
if(testMethods.size() > 0) {
  if(creator == null)
    try {
      if(!Modifier.isPublic(testClass
         .getDeclaredConstructor().getModifiers())) {
        print("Error: " + testClass +
          " default constructor must be public");
        System.exit(1);
      }
    } catch(NoSuchMethodException e) {
      // Costruttore predefinito sintetizzato; OK
    }
  print(testClass.getName());
}
for(Method m : testMethods) {
  printnb("  . " + m.getName() + " ");
  try {
    Object testObject = createTestObject(creator);
    boolean success = false;
    try {
      if(m.getReturnType().equals(boolean.class))
        success = (Boolean)m.invoke(testObject);
      else {
        m.invoke(testObject);
        success = true; // Senza assert non ha successo
      }
    } catch(InvocationTargetException e) {
      // L'eccezione reale e' in e:
      print(e.getCause());
    }
    print(success ? "" : "(failed)");
    testsRun++;
```

```
      if(!success) {
        failures++;
        failedTests.add(testClass.getName() +
          ": " + m.getName());
      }
      if(cleanup != null)
        cleanup.invoke(testObject, testObject);
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
}
static class TestMethods extends ArrayList<Method> {
  void addIfTestMethod(Method m) {
    if(m.getAnnotation(Test.class) == null)
      return;
    if(!(m.getReturnType().equals(boolean.class) ||
        m.getReturnType().equals(void.class)))
      throw new RuntimeException("@Test" method +
        " must return boolean or void");
    m.setAccessible(true); // Nel caso sia private, ecc.
    add(m);
  }
}
private static Method checkForCreatorMethod(Method m) {
  if(m.getAnnotation(TestObjectCreate.class) == null)
    return null;
  if(!m.getReturnType().equals(testClass))
    throw new RuntimeException("@TestObjectCreate " +
      "must return instance of Class to be tested");
  if((m.getModifiers() &
       java.lang.reflect.Modifier.STATIC) < 1)
    throw new RuntimeException("@TestObjectCreate " +
      "must be static.");
  m.setAccessible(true);
  return m;
```

```
  }
  private static Method checkForCleanupMethod(Method m) {
    if(m.getAnnotation(TestObjectCleanup.class) == null)
      return null;
    if(!m.getReturnType().equals(void.class))
      throw new RuntimeException("@TestObjectCleanup " +
        "must return void");
    if((m.getModifiers() &
        java.lang.reflect.Modifier.STATIC) < 1)
      throw new RuntimeException("@TestObjectCleanup " +
        "must be static.");
    if(m.getParameterTypes().length == 0 ||
       m.getParameterTypes()[0] != testClass)
      throw new RuntimeException("@TestObjectCleanup " +
        "must take an argument of the tested type.");
    m.setAccessible(true);
    return m;
  }
  private static Object createTestObject(Method creator) {
    if(creator != null) {
      try {
        return creator.invoke(testClass);
      } catch(Exception e) {
        throw new RuntimeException("Couldn't run " +
          "@TestObject (creator) method.");
      }
    } else { // Utilizza il costruttore predefinito:
      try {
        return testClass.newInstance();
      } catch(Exception e) {
        throw new RuntimeException("Couldn't create a " +
          "test object. Try using a @TestObject method.");
      }
    }
  }
} ///:~
```

**AtUnit.java** si serve dello strumento **ProcessFiles** presente in **net.mindview.util**. La classe **AtUnit** implementa **ProcessFiles.Strategy**, che contiene il metodo **process()**. In questo modo un'istanza di **AtUnit** può essere passata al costruttore di **ProcessFiles**. Il secondo argomento del costruttore ordina a **ProcessFiles** di cercare tutti i file con estensione "**class**".

Se non fornite un argomento da riga di comando il programma scorrerà l'alberatura di directory corrente; è consentito anche indicare argomenti multipli, che possono essere file di classe, con o senza estensione **.class**, oppure directory. Come si è detto, poiché **@Unit** individuerà automaticamente le classi e i metodi da testare, a differenza di JUnit non è necessario che i test siano raggruppati in suite.[8]

Uno dei problemi che **AtUnit.java** deve risolvere quando rileva i file di classe è che il nome qualificato della classe, comprendente cioè il nome del package, non è evidente dal semplice nome del file di classe. Per ottenere questa informazione, il file di classe deve essere analizzato, operazione certo non semplice ma neppure impossibile, come vedrete tra breve.[9]

Pertanto, quando viene rilevato un file **.class** innanzitutto viene aperto e i suoi dati binari vengono letti e passati a **ClassNameFinder.thisClass()**. A questo punto si entra nel mondo dell'"ingegneria del bytecode", poiché si sta analizzando il contenuto di un file di classe.

```
//: net/mindview/atunit/ClassNameFinder.java
package net.mindview.atunit;
import java.io.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class ClassNameFinder {
  public static String thisClass(byte[] classBytes) {
    Map<Integer,Integer> offsetTable =
      new HashMap<Integer,Integer>();
    Map<Integer,String> classNameTable =
      new HashMap<Integer,String>();
```

8. Non è chiaro perché il costruttore predefinito per la classe da testare debba essere **public**: qualora non lo sia, la chiamata **newInstance()** si interromperà senza sollevare un'eccezione.
9. Jeremy Meyer e l'autore hanno dedicato un intero giorno a progettare questo meccanismo.

```
try {
  DataInputStream data = new DataInputStream(
    new ByteArrayInputStream(classBytes));
  int magic = data.readInt();  // 0xcafebabe
  int minorVersion = data.readShort();
  int majorVersion = data.readShort();
  int constant_pool_count = data.readShort();
  int[] constant_pool = new int[constant_pool_count];
  for(int i = 1; i < constant_pool_count; i++) {
    int tag = data.read();
    int tableSize;
    switch(tag) {
      case 1: // UTF
        int length = data.readShort();
        char[] bytes = new char[length];
        for(int k = 0; k < bytes.length; k++)
          bytes[k] = (char)data.read();
        String className = new String(bytes);
        classNameTable.put(i, className);
        break;
      case 5: // LONG
      case 6: // DOUBLE
        data.readLong(); // scarta 8 byte
        i++; // Salto speciale necessario
        break;
      case 7: // CLASS
        int offset = data.readShort();
        offsetTable.put(i, offset);
        break;
      case 8: // STRING
        data.readShort(); // scarta 2 byte
        break;
      case 3:  // INTEGER
      case 4:  // FLOAT
      case 9:  // FIELD_REF
      case 10: // METHOD_REF
```

```
        case 11: // INTERFACE_METHOD_REF
        case 12: // NAME_AND_TYPE
          data.readInt(); // scarta 4 byte;
          break;
        default:
          throw new RuntimeException("Bad tag " + tag);
      }
    }
    short access_flags = data.readShort();
    int this_class = data.readShort();
    int super_class = data.readShort();
    return classNameTable.get(
      offsetTable.get(this_class)).replace('/', '.');
  } catch(Exception e) {
    throw new RuntimeException(e);
  }
}
// Dimostrazione:
public static void main(String[] args) throws Exception {
  if(args.length > 0) {
    for(String arg : args)
      print(thisClass(BinaryFile.read(new File(arg))));
  } else
    // Percorre l'intera alberatura:
    for(File klass : Directory.walk(".", ".*\\.class"))
      print(thisClass(BinaryFile.read(klass)));
}
} ///:~
```

Sebbene per ragioni di spazio non sia possibile analizzare in dettaglio l'intero codice, è evidente che ogni file di classe segue un formato particolare. L'autore ha cercato di utilizzare nomi di campo espressivi per le parti di dati ottenute dal flusso **ByteArrayInputStream**; potete notare anche le dimensioni delle varie porzioni di dati dalla lunghezza della lettura eseguita sul flusso in input: per esempio, i primi 32 bit di ogni file class sono rappresentati dal "numero magico" di valore esadecimale **hex 0xcafebabe**; i due valori **short** che seguono si riferiscono alle informazioni di versione.

La cosiddetta *constant pool* contiene le costanti del programma, pertanto è di dimensione variabile; il valore **short** successivo specifica queste dimensioni, allocando poi un array di dimensioni adeguate. Ogni voce nella constant pool può essere un valore a dimensione fissa o variabile: è quindi necessario esaminare il tag iniziale di ogni voce per capire come gestire la voce stessa. Questa operazione avviene nell'istruzione **switch**, che non ha lo scopo di analizzare con precisione tutti i dati presenti nel file di classe, bensì semplicemente di memorizzarne le parti rilevanti. Questo è il motivo per cui viene scartata una grande quantità di dati. Le informazioni sulle classi vengono memorizzate in **classNameTable** e in **offsetTable**. Dopo la lettura della "constant pool" sono disponibili le informazioni **this_class**: si tratta di un indice in **offsetTable** che produce un indice nella **classNameTable**, che a sua volta fornisce il nome della classe.

Quando il controllo ritorna a **AtUnit.java**, il metodo **process()** dispone del nome della classe e può così verificare se contiene un "**.**", che ne indica l'appartenenza a un package. Le classi che non sono parte di un package vengono ignorate, mentre tutte le altre vengono caricate, ricorrendo al caricatore delle classi standard, tramite il metodo **Class.forName()**; a quel punto la classe può essere analizzata alla ricerca delle annotazioni di **@Unit**.

I tre elementi da ricercare sono: i metodi **@Test**, memorizzati in una lista di **TestMethods**, e gli eventuali metodi contrassegnati da **@TestObjectCleanup** e **@TestObjectCreate**. Se esistono metodi **@Test** ne viene visualizzato il nome della classe, in modo che l'utente possa controllare che cosa sta accadendo; poi vengono eseguiti i test previsti. Per ogni test eseguito viene visualizzato il nome del metodo, cui fa seguito una chiamata a **createTestObject()** che utilizzerà il metodo **@TestObjectCreate**, se esiste, oppure ritornerà di nuovo al costruttore predefinito in caso contrario.

Una volta che l'oggetto di test è stato creato, su di esso viene invocato il metodo di test. Se questo restituisce un valore **boolean** (ipotesi 1) il risultato verrà intercettato, altrimenti, in assenza di eccezioni (ipotesi A) (che si verificherebbero in caso di errata **assert**) il programma darà per scontato che il test abbia avuto successo. Qualora vengano sollevate eccezioni (ipotesi B) le informazioni visualizzate permetteranno di comprenderne la causa. Se il test dovesse fallire (ipotesi 2), infine, il conteggio dei test falliti verrà incrementato e il nome e il metodo della classe "incriminati" verranno aggiunti a **failedTests**, e successivamente segnalati al termine dell'esecuzione.

**Esercizio 11** (5) Aggiungete un'annotazione **@TestNote** al framework **@Unit**, in modo che il test visualizzi una nota descrittiva.

## Rimozione del codice di test

Malgrado per molti progetti non faccia alcuna differenza lasciare il codice di test nel prodotto destinato al cliente, in particolare se tutti i metodi di test sono **private**, in alcuni casi potrebbe essere necessario eliminare tale codice per ridurre le dimensioni dei pacchetti o per evitare di fornire dettagli non richiesti.

Questo compito richiede un'ingegneria del bytecode molto più specializzata di quella che siete soliti impiegare manualmente, che tuttavia la libreria open source Javassist rende accessibile.[10]

Il programma mostrato di seguito accetta un'opzione **-r** come primo argomento. La presenza di questo flag rimuoverà le annotazioni **@Test**, mentre la sua assenza si limiterà a visualizzare le annotazioni **@Test**. In questo codice è utilizzato anche **ProcessFiles**, al fine di percorrere i file e le directory di vostra scelta.

```
//: net/mindview/atunit/AtUnitRemover.java
// Visualizza le annotazioni @Unit nei file class compilati. Se
// il primo argomento e' "-r", le annotazioni vengono
// eliminate.
// {Args: ..}
// {Requires: javassist.bytecode.ClassFile;
// Occorre installare la libreria Javassist scaricabile da
// http://sourceforge.net/projects/jboss/ }
package net.mindview.atunit;
import javassist.*;
import javassist.expr.*;
import javassist.bytecode.*;
import javassist.bytecode.annotation.*;
import java.io.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class AtUnitRemover
implements ProcessFiles.Strategy {
```

10. Un ringraziamento particolare va al Dr. Shigeru Chiba per la creazione di questa libreria e per l'aiuto fornito all'autore nello sviluppo di **AtUnitRemover.java**.

```
  private static boolean remove = false;
  public static void main(String[] args) throws Exception {
    if(args.length > 0 && args[0].equals("-r")) {
      remove = true;
      String[] nargs = new String[args.length - 1];
      System.arraycopy(args, 1, nargs, 0, nargs.length);
      args = nargs;
    }
    new ProcessFiles(
      new AtUnitRemover(), "class").start(args);
  }
  public void process(File cFile) {
    boolean modified = false;
    try {
      String cName = ClassNameFinder.thisClass(
        BinaryFile.read(cFile));
      if(!cName.contains("."))
        return; // Ignora le classi non pacchettizzate
      ClassPool cPool = ClassPool.getDefault();
      CtClass ctClass = cPool.get(cName);
      for(CtMethod method : ctClass.getDeclaredMethods()) {
        MethodInfo mi = method.getMethodInfo();
        AnnotationsAttribute attr = (AnnotationsAttribute)
          mi.getAttribute(AnnotationsAttribute.visibleTag);
        if(attr == null) continue;
        for(Annotation ann : attr.getAnnotations()) {
          if(ann.getTypeName()
             .startsWith("net.mindview.atunit")) {
            print(ctClass.getName() + " Method: "
              + mi.getName() + " " + ann);
            if(remove) {
              ctClass.removeMethod(method);
              modified = true;
            }
          }
        }
      }
```

```
      }
      // In questa versione i campi non vengono eliminati
      // (si veda il testo).
      if(modified)
        ctClass.toBytecode(new DataOutputStream(
          new FileOutputStream(cFile)));
      ctClass.detach();
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
} ///:~
```

La classe **ClassPool** è una sorta di rappresentazione di tutte le classi nell'applicazione che state modificando e garantisce la coerenza di tutte le classi modificate. Dovete ottenere ogni **CtClass** da **ClassPool**, in modo simile a come il caricatore di classi e **Class.forName()** caricano le classi nella JVM.

La classe **CtClass** contiene i bytecode di un oggetto di classe e consente di ottenere informazioni sulla classe e di manipolarne il codice. In questo caso viene chiamato il metodo **getDeclaredMethods()** (come il meccanismo di riflessione di Java), per ottenere da ogni oggetto **CtMethod** un oggetto **MethodInfo**, da cui potete esaminare le annotazioni. Se un metodo ha un'annotazione presente nel pacchetto **net.mindview.atunit**, viene rimosso.

Se la classe è stata modificata, il file **.class** originale viene sovrascritto con la nuova classe.

Al momento della redazione di questo manuale la funzionalità di rimozione in Javassist era un'aggiunta recente, e l'autore ha rilevato che l'eliminazione dei campi **@TestProperty** risultava più complessa della rimozione dei metodi.[11]

Poiché questi campi possono implicare operazioni di inizializzazione statiche non è possibile rimuoverli, di conseguenza la versione dell'esempio precedente elimina soltanto i metodi **@Unit**. Tuttavia è opportuno che controlliate il sito web di Javassist (*www.csg.is.titech.ac.jp/~chiba/javassist/*) per eventuali aggiornamenti: presto o tardi, la rimozione dei campi dovrebbe essere possibile. Nel frattempo ricordate che il metodo di test esterno mostrato in

11. Il Dr. Shigeru Chiba ha aggiunto la **CtClass.removeMethod()** su richiesta dell'autore.

**AtUnitExternalTest.java** consente la rimozione di tutti i test, semplicemente cancellando il file di classe creato dal codice di test.

## Riepilogo

Le annotazioni sono un'aggiunta molto gradita di Java, un meccanismo strutturato e dotato di controllo dei tipi, per l'aggiunta di metadati al vostro codice, che evita di renderlo illeggibile. Le annotazioni possono contribuire a eliminare l'onere di scrivere descrittori di implementazione e la necessità di generare altri file. Il fatto che il tag **@deprecated** di Javadoc sia stato sostituito dall'annotazione **@Deprecated** è un'indicazione di quanto le annotazioni siano più adatte per descrivere le informazioni sulle classi di quanto non lo fossero i commenti.

Java SE5 è distribuito soltanto con pochissime annotazioni predefinite: questo significa che, se non trovate una libreria adatta altrove, dovrete creare le annotazioni e la logica a esse associata. Con lo strumento **apt** potete compilare i file appena generati in un'unica soluzione, semplificando il processo di compilazione; attualmente, però, le API **mirror** non offrono che poche funzionalità di base per aiutarvi a identificare gli elementi delle definizioni delle classi Java. Come avete visto, potete servirvi di Javassist per l'ingegneria del bytecode, o se preferite potete scrivere voi stessi le utility per la manipolazione del bytecode.

Questa situazione certamente migliorerà, e i produttori di API e di framework inizieranno a integrare le annotazioni nel software da loro fornito. Come potete intuire esaminando il framework **@Unit**, è molto probabile che le annotazioni provocheranno cambiamenti significativi nell'esperienza comune dei programmatori Java.

*La soluzione degli esercizi è disponibile nel documento The Thinking in Java Annotated Solution Guide, in vendita all'indirizzo www.mindview.net.*

# Appendice A
# Supplementi

A integrazione di questo volume sono disponibili alcuni supplementi, tra cui la documentazione, i seminari e i servizi offerti sul sito web di MindView.

Questa appendice è la presentazione di tali supplementi, che vi consentirà di valutare se potranno esservi di aiuto.

Tenete presente che, anche se la maggior parte dei seminari è stata pensata per un pubblico vasto, potete richiedere corsi di addestramento privati e seminari aziendali presso la vostra sede.

## Supplementi scaricabili

Il codice di questo volume è scaricabile all'indirizzo *www.mindview.net*. Si tratta fondamentalmente dei file di build Ant e di altri file di supporto necessari per costruire ed eseguire tutti gli esempi presentati nel libro.

Alcune parti del volume sono state inoltre convertite in formato elettronico, in particolare i seguenti argomenti:

**1.** clonazione di oggetti (*Cloning Objects*);

**2.** passaggio e restituzione di oggetti (*Passing & Returning Objects*);

3. analisi e progettazione (*Analysis and Design*);
4. porzioni di altri capitoli tratti da *Thinking in Java, terza edizione*, che l'autore non ha ritenuto appropriato includere nella versione cartacea della quarta edizione di questo testo.

## Thinking in C: i fondamenti di Java

Sul sito *www.mindview.net* potete scaricare gratuitamente il seminario *Thinking in C*: questa presentazione, creata da Chuck Allison e realizzata da MindView, è un corso multimediale in formato Flash che vi fornirà le nozioni introduttive alla sintassi, agli operatori e alle funzioni del linguaggio C, sui quali si basa la sintassi Java.

Tenete presente che l'utilizzo di questo seminario richiede l'installazione del software Flash Player di *www.macromedia.com*.

## Seminari Thinking in Java

La società dell'autore, MindView, Inc., eroga seminari e corsi di istruzione teorico-pratici di cinque giorni, aperti al pubblico e ai privati, in cui si sviluppano gli argomenti trattati in questo volume. Noto in precedenza come *Hands-On Java*, questo è il principale corso introduttivo che fornisce le nozioni necessarie per accedere agli altri seminari avanzati di MindView. Ogni lezione è realizzata con materiale selezionato da ogni capitolo ed è seguita da un periodo di esercitazioni controllate, in modo tale che ogni partecipante riceva la necessaria attenzione. Per informazioni su calendario, sedi dei corsi, testimonianze e ulteriori dettagli, consultate il sito *www.mindview.net*.

## Seminario Hands-On Java su CD

Il CD Hands-On Java contiene una versione ampliata del seminario *Thinking in Java*, ed è anch'esso basato sui contenuti di questo libro; è stato concepito per fornire alcune delle esperienze del corso *live*, senza dovere sostenere l'onere del viaggio e del soggiorno. Il CD si compone di letture audio e diapositive corrispondenti ai diversi capitoli del libro. L'autore ha realizzato personalmente il seminario: il materiale è in formato Flash, adatto quindi a qualunque piattaforma supporti il software Flash Player. Il CD *Hands-On Java* è in vendita sul sito *www.mindview.net*, dove potrete scaricare anche versioni dimostrative di questo prodotto.

## Seminario Thinking in Objects

Questo corso introduce i concetti della programmazione a oggetti dal punto di vista del progettista, esplorando il processo di sviluppo e costruzione di un'applicazione, con particolare attenzione alle cosiddette "metodologie leggere" (*Agile Method* o *Lightweight Methodology*) e alla "programmazione estrema" (*XP, Extreme Programming*). Vi saranno illustrate le metodologie in generale, strumenti di utilità quali le tecniche di pianificazione *index-card* (descritte in *Planning Extreme Programming* di Beck e Fowler, Addison-Wesley, 2001), le schede CRC per la progettazione a oggetti, la programmazione "in coppie" (*pair programming*), l'*iteration planning*, i test unitari, il building automatizzato, il controllo del codice sorgente e argomenti analoghi. Il corso comprende un progetto XP, che verrà sviluppato durante la settimana formativa.

Se state per avviare un progetto e intendete usufruire di tecniche di progettazione orientate agli oggetti, potrete utilizzare le vostre tematiche come esempio e produrre una prima bozza di design nel corso della formazione.

Per informazioni su calendario, sedi dei corsi, testimonianze e ulteriori dettagli consultate il sito *www.mindview.net*.

## Thinking in Enterprise Java

Questo libro ha avuto origine da alcuni dei capitoli più elaborati delle precedenti edizioni di *Thinking in Java*: non è una sorta di "secondo volume" di *Thinking in Java*, bensì un'attenta analisi degli argomenti più avanzati della programmazione d'impresa. Sebbene sia ancora in fase di sviluppo, attualmente è già disponibile per il download gratuito dal sito *www.mindview.net*. Trattandosi di un libro separato, sarà possibile una sua successiva espansione per l'adeguata trattazione degli argomenti richiesti. Come per *Thinking in Java*, l'obiettivo è fornire un'introduzione agli elementi di base delle tecnologie di programmazione d'impresa, in modo che il lettore sia preparato ad affrontare in dettaglio questa materia. Tra gli argomenti citiamo i seguenti:

1. introduzione alla programmazione d'impresa;
2. programmazione di rete con socket e canali;
3. chiamate RMI (*Remote Method Invocation*);
4. connessione ai database;
5. servizi di rete (*Naming Service* e *Directory Service*);
6. servlet;

7. pagine JSP (*Java Server Page*);
8. tag, frammenti JSP e linguaggio di espressione;
9. automazione della creazione di interfacce utente;
10. enterprise JavaBeans;
11. XML;
12. servizi web;
13. test automatici.

Potete scaricare la versione corrente di *Thinking in Enterprise Java* all'indirizzo *www.mindview.net*.

## Thinking in Patterns (with Java)

Uno dei passi fondamentali nella progettazione orientata agli oggetti è stato il cosiddetto "movimento dei design pattern", riassunto nel volume *Design Patterns*, di Gamma, Helm, Johnson e Vlissides (Addison-Wesley, 1995). Questo libro presenta 23 classi di problemi generici e le relative soluzioni, scritte soprattutto in C++, ed è un testo essenziale per un argomento che oggi è ormai quasi obbligatorio per qualunque programmatore OOP. *Thinking in Patterns* introduce i concetti di base di questi "modelli di progettazione", nonché alcuni esempi realizzati in Java: il libro non vuole essere una mera traduzione di *Design Patterns* in un diverso linguaggio, bensì una nuova prospettiva dall'ottica di Java; non si limita inoltre ai 23 modelli tradizionali, ma include altri concetti e tecniche per la soluzione dei problemi.

Il libro è nato come capitolo finale di *Thinking in Java, prima edizione*, ma assistendo al continuo sviluppo dei concetti è apparso chiaro che sarebbe dovuto diventare un volume autonomo. Al momento della stesura di queste note, il testo è ancora in fase di elaborazione: il materiale di base è stato più volte aggiornato con numerose presentazioni del seminario *Objects & Patterns*, ora ristrutturato nei due corsi *Designing Objects & Systems* e *Thinking in Patterns*.

Troverete maggiori dettagli su questo libro all'indirizzo *www.mindview.net*.

## Seminario Thinking in Patterns

Questo corso è un'evoluzione del precedente seminario *Objects & Patterns* tenuto da Bill Venners e dall'autore negli ultimi anni. Considerato il progressivo aumento del numero di argomenti, si è ritenuto opportuno suddividerlo in due parti: *Thinking in Patterns*, appunto, e *Designing Objects & Systems*.

Il seminario ricalca fedelmente il materiale e lo schema di presentazione del volume *Thinking in Patterns*, pertanto il modo migliore per conoscere i contenuti del corso è consultare i dettagli sul libro disponibili sul sito *www.mindview.net*.

La presentazione è focalizzata soprattutto sul processo di evoluzione della progettazione, partendo da una soluzione iniziale e procedendo gradualmente nell'evoluzione logica e procedurale verso soluzioni progettuali più appropriate. L'ultimo progetto illustrato, la simulazione di un impianto per il riciclaggio dei rifiuti, si è evoluto nel corso degli anni: questo vi permette di considerare tale miglioramento come prototipo del modo in cui i vostri progetti, inizialmente semplici soluzioni a problemi specifici, potranno evolversi in un approccio più flessibile a un'intera categoria di problematiche.

Questo seminario vi aiuterà a:

1. potenziare in modo notevole la flessibilità dei vostri progetti;
2. implementare i concetti di estensibilità e di "riutilizzabilità" (*extensibility & reusability*);
3. incrementare il flusso di comunicazioni tra progetti, ricorrendo al linguaggio dei pattern.

Al termine di ogni incontro i partecipanti riceveranno un insieme di esercizi-modello da risolvere, nei quali saranno guidati alla realizzazione di codice che applica pattern specifici alla soluzione dei problemi di programmazione.

Per informazioni su calendario, sedi dei corsi, testimonianze e ulteriori dettagli, consultate il sito *www.mindview.net*.

## Consulenza e revisione di progetti

MindView mette a disposizione servizi di consulenza, tutoraggio, revisione e applicazioni a supporto del ciclo di sviluppo dei vostri progetti, anche per il primo progetto Java della vostra azienda. Per conoscere disponibilità e dettagli dei servizi, consultate il sito *www.mindview.net*.

# Appendice B

# Risorse

## Software

Il **JDK**, disponibile all'indirizzo *http://java.sun.com*. Anche se deciderete di utilizzare un ambiente di sviluppo di altri produttori, è sempre opportuno che abbiate a disposizione JDK, soprattutto nel caso vi troviate alle prese con un errore di compilazione. JDK è la pietra di paragone, il punto di riferimento in cui certamente troverete segnalato ogni possibile bug.

**Documentazione JDK** all'indirizzo *http://java.sun.com*, in formato HTML. È quasi impossibile trovare un testo sulle librerie standard di Java che non sia obsoleto o mancante di informazioni.

Malgrado la documentazione JDK di Sun presenti qualche bug e talvolta sia sintetica al punto da essere quasi inutilizzabile, ha il vantaggio di elencare *tutti* i metodi e le classi.

Alcuni utenti hanno inizialmente difficoltà a utilizzare una risorsa online invece di un testo cartaceo: è bene tuttavia che vi abituiate a servirvi dei documenti HTML, in modo da disporre della panoramica completa della situazione; in caso di difficoltà, potrete sempre ripiegare su un testo stampato.

## Editor e ambienti IDE

In questo settore la competizione è davvero agguerrita. Molte soluzioni sono gratuite e di solito quelle a pagamento offrono versioni dimostrative, pertanto non dovrete far altro che provarle e verificare se qualcuna corrisponde alle vostre esigenze. Ecco alcune proposte:

**JEdit**, l'editor gratuito di Slava Pestov, è scritto in Java, quindi offre il vantaggio aggiuntivo di mostrarvi in azione un'applicazione desktop scritta in questo linguaggio. JEdit è supportato da un gran numero di plugin, molti dei quali scritti dalla comunità open source. Il software può essere scaricato all'indirizzo *www.jedit.org*.

**NetBeans** è un ambiente IDE gratuito di Sun, disponibile all'indirizzo *www.netbeans.org*. Tra le funzionalità fornite, occorre segnalare la possibilità di "disegnare" grafiche mediante trascinamento, posizionamento e ridimensionamento, l'editing del codice e il debugging.

**Eclipse**, un progetto open source sostenuto, tra gli altri, da IBM. La piattaforma Eclipse è stata progettata con criteri di estensibilità, per consentirvi di produrre applicazioni standalone basate su questa piattaforma. È con questo software che è stata creata la libreria SWT (*Standard Widget Toolkit*) descritta nel Volume 3, Capitolo 2. Il software Eclipse può essere scaricato dal sito *www.eclipse.org*.

**IDEA di IntelliJ** è il software preferito di un gran numero di programmatori Java, molti dei quali ritengono che IDEA sia sempre un passo avanti rispetto a Eclipse, forse per il fatto che IntelliJ non offre contemporaneamente un'interfaccia IDE e una piattaforma di sviluppo, ma si limita a fornire l'ambiente IDE. Potete scaricare una versione dimostrativa di IDEA dal sito *www.jetbrains.com*.

## Libri

***Effective Java™ Programming Language Guide***, di Joshua Bloch (Addison-Wesley, 2001). Un testo che non deve mancare nella vostra libreria tecnica.

***Core Java™ 2, settima edizione, Volumi I e II***, di Horstmann & Cornell (Prentice Hall, 2005). Un testo ricco e completo, al quale molti ricorrono come risorsa di riferimento. L'autore raccomanda questo libro a chi desideri approfondire i concetti presentati in *Thinking in Java*.

***The Java™ Class Libraries: An Annotated Reference***, di Patrick Chan e Rosanna Lee (Addison-Wesley, 1997). Benché obsoleto, questo libro contiene ciò che dovrebbe offrire JDK, vale a dire una descrizione sufficiente a rende-

re utilizzabile il software. Uno dei revisori di *Thinking in Java* ha affermato: "Se dovessi dare la preferenza a un solo titolo su Java sceglierei questo". L'autore non è della stessa opinione, poiché ritiene che il volume sia troppo imponente (e costoso) e che la qualità degli esempi non sia soddisfacente. Rimane *comunque* uno dei testi da consultare a fronte di problemi, e in molti casi affronta gli argomenti in modo più approfondito di gran parte dei titoli alternativi. Rimane il fatto che ***Core Java™ 2*** esamina componenti di libreria molto più recenti.

***Java Network Programming, seconda edizione***, di Elliotte Rusty Harold (O'Reilly, 2000). L'autore ha dichiarato di avere sempre avuto difficoltà a comprendere l'argomento delle reti Java (delle reti in generale, a dire il vero), finché non ha scoperto questo titolo. Anche il sito web di supporto, Café au Lait (*www.cafeaulait.org*), offre punti di vista stimolanti, opinioni notevoli e aggiornamenti accurati sullo sviluppo Java, non vincolati ad alcun produttore. La puntualità degli aggiornamenti vi permetterà di rimanere al passo con il mondo Java, sempre in movimento.

***Design Patterns***, di Gamma, Helm, Johnson e Vlissides (Addison-Wesley, 1995). Questo è il testo fondamentale che ha dato l'avvio al movimento della programmazione per modelli, di cui si parla diffusamente in *Thinking in Java*.

***Refactoring to Patterns***, di Joshua Kerievsky (Addison-Wesley, 2005). Una perfetta combinazione di refactoring e design pattern. L'aspetto più interessante di questo libro è che vi mostra come fare evolvere un progetto adattando i modelli alle vostre esigenze.

***The Art of UNIX Programming***, di Eric Raymond (Addison-Wesley, 2004). Malgrado Java sia un linguaggio multipiattaforma, la sua prevalenza su sistemi server ha dimostrato quanto sia importante la conoscenza dell'ambiente Unix/Linux. Questo testo è un'eccellente introduzione storica e filosofica a tale sistema operativo, una lettura avvincente anche per chi voglia soltanto conoscere alcuni aspetti dell'informatica.

## Analisi e progettazione

***Extreme Programming Explained, seconda edizione***, di Kent Beck e Cynthia Andres (Addison-Wesley, 2005). L'autore ha sempre sostenuto che dovrebbe esistere un modo migliore per affrontare lo sviluppo dei programmi, e XP sembra essere l'approccio giusto. L'unico libro che molti considerano di pari livello è *Peopleware*, descritto di seguito, che affronta principalmente l'ambiente e la cultura aziendale. *Extreme Programming Explained* tratta di programmazione e trasforma la maggior parte degli argomenti, anche le "scoperte" più recenti, in temi sorprendenti ed eccitanti. Gli autori di que-

sto libro arrivano addirittura a sostenere che le illustrazioni sono accettabili purché non si dedichi loro troppo tempo, tanto che sarebbero propensi a eliminarle (noterete infatti che il libro *non* ha il contrassegno di approvazione "UML stamp of approval").

Pensate come deve essere accettare di lavorare per una software house soltanto se questa adotta i principi della programmazione XP... Libri brevi, capitoli concisi e facili da leggere, idee entusiasmanti sulle quali riflettere. Iniziate a immaginare di lavorare in una simile atmosfera, che vi introdurrà in anteprima a un mondo completamente nuovo.

***UML Distilled, seconda edizione***, di Martin Fowler (Addison-Wesley, 2000). Avvicinandovi a UML potreste rimanerne intimiditi, a causa dei numerosi diagrammi e dettagli che caratterizzano questo linguaggio. Fowler ritiene che gran parte di questa sovrastruttura sia ridondante, pertanto la riduce all'essenziale. Per quasi tutti i progetti è sufficiente la padronanza di poche utility grafiche, e l'intento di Fowler è soprattutto quello di insegnarvi a realizzare un buon progetto senza soffermarsi eccessivamente sui dettagli: in effetti, di norma la lettura della sola prima parte del libro è più che adeguata. *UML Distilled* è un buon libro, non voluminoso, di cui apprezzerete la lettura: il primo che dovreste procurarvi per comprendere il linguaggio UML.

***Domain-Driven Design***, di Eric Evans (Addison-Wesley, 2004). Questo libro considera il modello di dominio (*domain model*) come strumento primario nel processo di progettazione. L'autore ritiene che questa sia una svolta determinante nella visione progettuale, in grado di portare i progettisti a conseguire un livello di astrazione ottimale.

***The Unified Software Development Process***, di Ivar Jacobsen, Grady Booch e James Rumbaugh (Addison-Wesley, 1999). L'autore era prevenuto nei confronti di questo libro, che sembrava avere tutti i presupposti del tipico, noiosissimo testo universitario. In realtà si è rivelato una piacevole sorpresa, anche se in alcuni punti sembra che i concetti non siano molto chiari neppure agli autori. Nell'insieme, tuttavia, il libro non solo risulta scorrevole, ma anche ben scritto e, soprattutto, l'intero processo è illustrato in modo pratico e perfettamente coerente. Non si tratta di programmazione estrema (XP), perché non ne possiede la chiarezza in termini di test, ma è anch'essa parte dell'inesorabile valanga UML; una forza che, sebbene non avvicini direttamente gli utenti alla programmazione XP, ha fatto sì che molte persone, a prescindere dall'effettivo livello di esperienza con questa tecnologia, abbracciassero la "causa UML". L'autore ritiene che questo libro dovrebbe diventare il portabandiera del linguaggio UML, un testo che potrete leggere dopo aver affrontato *UML Distilled* di Fowler, qualora vogliate approfondire UML nei dettagli.

Prima di adottare una metodologia, è sempre utile conoscere il punto di vista di chi non si propone di vendervi nulla. È facile adottare una tecnologia senza comprendere appieno le aspettative che si hanno su di essa, o le caratteristiche che questa può offrire; per molti, è sufficiente il fatto che altri la utilizzino. Tuttavia alcuni hanno una curiosa perversione: se sono convinti che qualcosa risolverà i loro problemi, faranno di tutto per provarla (la *sperimentazione*, un approccio positivo); ma non appena si renderanno conto che i loro problemi non vengono risolti, raddoppieranno gli sforzi e annunceranno al mondo di avere scoperto qualcosa di eccezionale (questo significa *rinnegare*, un approccio tutt'altro che positivo). Molti ritengono che sia sufficiente fare salire sulla propria barca più gente possibile soltanto per non sentirsi soli, anche se la barca non ha una rotta precisa o se è destinata ad affondare.

Con ciò, l'autore non vuole affatto affermare che tutte le metodologie conducano al nulla: dovrete semplicemente armarvi fino ai denti di strumenti mentali che vi aiuteranno a rimanere su un livello di sperimentazione ("Non funziona, proviamo qualcos'altro"), ben lontani dalla condizione di "rifiuto" ("No, non è un problema: è tutto a posto, non c'è niente da cambiare"). L'autore ritiene che i titoli elencati di seguito, letti prima della fase di adozione di un metodo, possano fornirvi questi strumenti mentali.

***Software Creativity***, di Robert L. Glass (Prentice Hall, 1995). Secondo l'autore questo è il libro migliore che analizza le prospettive sull'intero problema delle metodologie: è una raccolta di saggi brevi e articoli che Glass ha scritto e in alcuni casi acquisito (P. J. Plauger è uno dei collaboratori), il risultato di molti anni di studio sull'argomento. Interventi piacevoli e lunghi a sufficienza da esprimere i concetti necessari: l'autore non vi annoierà né affliggerà con vaneggiamenti di sorta. Né tantomeno tenterà di vendere fumo, dato che propone concetti suffragati da centinaia di riferimenti ad altri articoli e saggi. Tutti i programmatori e i manager dovrebbero dedicarsi a questa lettura prima di guadare il pantano delle metodologie.

***Software Runaways: Monumental Software Disasters***, di Robert L. Glass (Prentice Hall, 1998). Il pregio di questo libro è che pone in risalto ciò di cui non si parla mai: la quantità di progetti che non soltanto falliscono, ma lo fanno in modo spettacolare. Molti pensano che a loro non potrà mai accadere nulla di simile, o quantomeno che non potrà *più* succedere, e questo è un atteggiamento sbagliato: se terrete sempre presente che le cose possono andare male, sarete sempre nella posizione giusta per poter migliorarle.

***Peopleware, seconda edizione***, di Tom DeMarco e Timothy Lister (Dorset House, 1999). Un titolo che tutti *dovrebbero* leggere: non soltanto è divertente, ma è un vero terremoto che metterà a soqquadro il vostro mondo e distruggerà

i vostri preconcetti. Sebbene DeMarco e Lister provengano da un'esperienza di sviluppo software, il libro prende in esame i progetti e i team operativi in generale, ma mettendo a fuoco le *persone* e le loro necessità, non tanto la tecnologia e le sue esigenze. In *Peopleware* vedrete come creare un ambiente in cui le persone siano soddisfatte e produttive, senza il solito elenco di regole a cui occorre conformarsi per diventare perfetti "componenti" di una macchina: un'attitudine, quest'ultima, che spesso è causa di un falso atteggiamento da parte di molti programmatori, i quali aderiscono con finto entusiasmo a ogni nuovo metodo X, e poi tornano a fare esattamente come prima.

***Secrets of Consulting: A Guide to Giving & Getting Advice Successfully***, di Gerald M. Weinberg (Dorset House, 1985). Un libro superbo, uno dei preferiti dell'autore, perfetto per chi si avvicina alla professione di consulente o per chi, come consulente, voglia progredire nella professione. Capitoli brevi, integrati con storie e aneddoti che illustrano come arrivare al nocciolo del problema con il minimo sforzo. Altri testi interessanti sull'argomento sono *More Secrets of Consulting*, pubblicato nel 2002, e molti altri lavori di Weinberg.

***Complexity***, di M. Mitchell Waldrop (Simon & Schuster, 1992). Questo libro è la cronaca dell'incontro che un gruppo di scienziati di varie discipline ha tenuto a Santa Fé, New Mexico, allo scopo di discutere problemi reali che le singole discipline non sono in grado di risolvere: il mercato azionario in economia, l'origine della vita in biologia, le motivazioni comportamentali in sociologia ecc. Dalla combinazione di fisica, economia, chimica, matematica, informatica, sociologia e altre scienze si sta sviluppando un approccio multidisciplinare ai problemi, ma soprattutto sta emergendo una *forma mentis* alternativa nei confronti di queste tematiche ultra-complesse: tale approccio è ben distante dal determinismo matematico e dall'illusione che un'equazione basti a predire qualsiasi comportamento, ma è volto all'osservazione preliminare, alla ricerca di un modello e al tentativo di emularlo a tutti i costi (nel libro è descritto, per esempio, l'emergere degli algoritmi genetici). Secondo l'autore, questo è un approccio prezioso per affrontare la gestione di progetti software sempre più complessi.

## Python

***Learning Python, seconda edizione***, di Mark Lutz e David Ascher (O'Reilly, 2003). Una valida introduzione al linguaggio preferito dell'autore, oltre che un eccellente compagno per Java. Il libro include un'introduzione a Jython, che vi consente di combinare Java e Python in un unico programma: l'interprete Jython è compilato in puro bytecode Java, quindi non vi serve niente di speciale nel caso dobbiate utilizzarlo. Un'unione di due linguaggi che preannuncia grandi possibilità.

## Bibliografia dell'autore

Nessuno di questi testi è più a catalogo, ma potreste trovarne alcuni nelle librerie specializzate.

***Computer Interfacing with Pascal & C*** (pubblicato in proprio per i tipi di Eisys, 1988. Disponibile in vendita soltanto presso *www.mindview.net*). Un'introduzione all'elettronica, che risale ai tempi in cui CP/M era ancora re e DOS muoveva i primi passi: per pilotare vari progetti elettronici, l'autore si è servito di linguaggi di alto livello e spesso della porta parallela del computer. Il libro è un adattamento degli articoli scritti dall'autore per la prima (e migliore) rivista con cui ha collaborato, Micro Cornucopia, che purtroppo ha cessato le pubblicazioni molto tempo prima che Internet diventasse popolare. La redazione di questo libro è stata una delle esperienze editoriali più soddisfacenti per l'autore.

***Using C++*** (Osborne/McGraw-Hill, 1989). Uno dei primi libri su C++, ormai fuori catalogo e sostituito dalla seconda edizione, intitolata *C++ Inside & Out*.

***C++ Inside & Out*** (Osborne/McGraw-Hill, 1993). Si tratta della seconda edizione di *Using C++*. Il codice C++ esaminato in questo libro è ragionevolmente accurato, ma risale al 1992 circa, e per questo motivo è stato sostituito da *Thinking in C++*. Troverete maggiori dettagli sul sito *www.mindview.net*, dove potrete anche scaricare il codice sorgente.

***Thinking in C++, prima edizione*** (Prentice Hall, 1995). Questo libro ha vinto il premio *Software Development Magazine Jolt Award* quale miglior libro dell'anno.

***Thinking in C++, seconda edizione, Volume 1*** (Prentice Hall, 2000). Scaricabile da *www.mindview.net* e aggiornato secondo gli standard finalizzati del linguaggio.

***Thinking in C++, seconda edizione, Volume 2***, scritto in collaborazione con Chuck Allison (Prentice Hall, 2003): il libro è scaricabile dal sito *www.mindview.net*.

***Black Belt C++: The Master's Collection***, a cura di Bruce Eckel (M&T Books, 1994). Questo libro, fuori catalogo, è una raccolta di testi di varie "autorità" in materia di C++, basati sulle loro presentazioni alla conferenza Software Development Conference, presieduta dall'autore. È stata la copertina di questo libro a stimolare Eckel nella progettazione di tutte le successive copertine dei suoi lavori.

***Thinking in Java, prima edizione*** (Prentice Hall, 1998). La prima edizione di questo libro ha vinto i premi *Software Development Magazine Productivity*

*Award, Java Developer's Journal Editor's Choice Award* e *JavaWorld Reader's Choice Award* come miglior libro. Questa edizione può essere scaricata dal sito *www.mindview.net*.

***Thinking in Java, seconda edizione*** (Prentice Hall, 2000). Questa edizione ha vinto il premio *JavaWorld Editor's Choice Award* quale miglior libro, ed è scaricabile da *www.mindview.net*.

***Thinking in Java, terza edizione*** (Prentice Hall, 2003). Questa edizione ha vinto il premio *Software Development Magazine Jolt Award* quale miglior libro dell'anno e numerosi altri premi, elencati in quarta di copertina. Il testo è scaricabile dal sito *www.mindview.net*.

# Indice analitico


**A**

accoppiamento (*coupling*) 128
annotazioni 641–696
  assenza di supporto dell'ereditarietà nelle 654
  definizione delle 643–645
  elaborazione delle, con apt 658–664
  elementi delle 648
  test unitari basati sulle 668–681
AOP (*Aspect-Oriented Programming*) 250
API Preferences 581–583
apt
  per l'elaborazione delle annotazioni 658–664
  utilizzo del modello Visitor con 664–668
archivi Java 548–550
argomenti variabili
  elenchi di 160–161
array 289–338
  bidimensionali, utilizzo di 638–639
  come oggetti di prima classe 292–295
  confronto di 324–325
  confronto tra elementi di 325–330
  copia di 322–323
  creazione di, con i generatori 315–321
  e generici 303–306
  multidimensionali 298–304
  ordinamento di 330–331
  ordinati, ricerca in 331–334
  restituzione degli 296–297
Arrays, classe 321
  utility per 321–333
Arrays.fill() 307–308
attributi
  aggiunta di 485–486
autolimitazione 238–241

**B**

backdoor 136
BASIC 2
big endian, approccio 528–529
BitSet 459–462
buffer
  di vista (*view buffer*) 524
  in dettaglio 531–534
  per la manipolazione dei dati 529–531
  per la visualizzazione 524–529
bufferizzazione dei file di input 492–494

**C**

C++ 141
cancellazione 177–179

compensazione della 191–193
problema della 184–186
canonicalized mapping 453
capacità (capienza) 531
capienza (*capacity*) 388
caratteri di nuova riga (*newline*) 493
caricatore di classe (*class loader*) 71
caricatore di classi principale
(*primordial class loader*) 71
cast, nuova sintassi di 84–85
casting
di tipo sicuro (*type-safe casting*) 85
e avvertimenti
del compilatore 231–234
vantaggi e svantaggi del 285–288
catenamento esterno (*external chaining*) 404
Chain of Responsibility, design
pattern 614
chiavi deboli (*weak key*) 388
Class, classe 108
Class, oggetto 70–77
equivalenza con
instanceof 105–107
classe di supporto (*companion class*) 341
classi 171
astratte, utilizzo delle 349–360
interne anonime 171–173, 468–471
non modificate 491
Reader 488–489
recupero di 556–557
riferimenti generici di 80–84
Writer 488–489
cleanup, esecuzione del,
con finally 32–40
code 379–384
con priorità 381–382
deque 383–384
codice di test,
rimozione del 693–696
codici hash 394–399
coefficiente di carico
(*load factor*) 388
Collection
funzionalità di 360–363
rendere immutabile una 446–448
sincronizzazione di 448–450
compatibilità,
per la migrazione 182–184
compilatore 212–214
compressione 543–550
semplice con GZIP 544–545
concatenamento di eccezioni 25
constant pool 692
conteggio ricorsivo 98–100
contenitori 339–464
di Java 1.0 e Java 1.1 455–462
popolamento dei 341–360
tassonomia completa dei 339–341
contrassegno (*token*) 163
controlli pre-casting 85–94
controvarianza 215–218
conversione dell'intercettazione
(*capture conversion*) 226
copia
byte-per-byte 323
shallow 323
costanti
metodi specifici per 609–627
utilizzo dei metodi
specifici per 634–637
costruttori 46–52
covarianza degli
argomenti 241–246
creazione di archivi di,
con ZIP 546–548

## D


dati
  conversione dei 518–522
  di prova, creazione dei 306–321
  generatori di 308–315
  manipolazione dei, tramite buffer 529–531
  memorizzazione e recupero dei 498–500
  origini e destinazioni dei 489–490
  tipi primitivi, ottenimento dei 522–524
Decorator, design pattern 481
  utilizzo di 254–256
deduzione del tipo
  utilizzo della 157–159
deque (coda doppia) 383–384
directory
  ottenere l'elenco di 466–468
  utility per 471–478
dispatching
  con enum 631–634
  con le EnumMap 637–638
double dispatching 637
downcast 85
DTO (*Data Transfer Object*) 345, 417
dynamic typing 60


## E


eccezioni 247–250
  argomenti delle 5
  concatenamento di 25–30
  controllate (*checked exception*) 17
  conversione delle, da controllate a non controllate 62–65
  corrispondenza delle 53–54
  di base 3–5
  e log 11–16
  gestori di 6–8
  guida di riferimento alle 65–66
  intercettazione delle 5, 17–29
  Java standard 29–30
  limiti delle 42–46
  non controllate (*unchecked exception*) 31
  passaggio delle alla console 61–62
  per la gestione degli errori 1–66
  personalizzate 8–11
  sollevate ripetutamente 21–25
elaboratori delle annotazioni (*annotation processor*) 644
  scrittura di 646–648
enum 585
  aggiunta di metodi a 588–591
  catena di responsabilità 614–620
  dispatching con 631–634
  funzionalità di base di 585–588
  macchine a stati con 620–627
  nelle dichiarazioni switch 591–592
  sovrascrittura dei metodi di 590–591
  utilizzo delle importazioni static 587–588
enumeration (enumerazione) 456
Enumeration, classe 456–458
EnumMap, utilizzo di 608–609
EnumSet, utilizzo di, in alternativa ai flag 605–608
errori, gestione dei 1–66
estrattore dei metodi di classe 109–112
exception handler 3
exception handling 2
Externalizable, interfaccia
  alternativa a 565–568

F

Factory Method, design pattern 101
factory registrate 100–105
factory registrate (*Registered Factory*) 101
fail-fast, meccanismo del 449–450
file
  ad accesso casuale, lettura e scrittura di 501–503
  binari, lettura dei 507–508
  blocco dei 540–543
  di output, nozioni fondamentali 496–497
  esterni, generazione di 649–654
  mappati, blocco parziale dei 541–543
  mappati in memoria 535–539
  utility per la scrittura e la lettura 503–507
File, classe 466–480
file descrittori (*descriptor file*) 641
filtro di directory (*directory filter*) 466
finally 32–40
  funzione di 34–37
  utilizzo di, durante un return 38–39
flussi (*stream*) 481
  con pipe 503
  convogliati (*piped stream*) 503
  modifica del comportamento dei 490–491
  utilizzi tipici dei 492–503
Flyweight, design pattern 349
framework 416
funzione di hash perfetta 403

G

Generator 342–345
  multiutilizzo 162–164
generici 139–288
  array di 197–203
  problemi relativi ai 227–236
  semplici 141–143
  stranamente ricorsivi 236–238
gestione degli errori, con le eccezioni 1–66
gestore delle eccezioni (*exception handler*) 6
  tecniche alternative 54–56
gestore di chiamate (*invocation handler*) 114
guarded region 5
GUI (*Graphical User Interface*) 107

H

hashCode()
  approfondimento su 399–402
  sovrascrittura di 408–415
hashing 394–399
  ottimizzazione del 403–408
HashMap, fattori che influenzano le prestazioni di 438–439
Hashtable 457

I

IDE (*Integrated Development Enrivonment*) 107
implementazione 596–597
  scelta della 415–439
inferenza degli argomenti di tipo (*type argument inference*) 157
informazioni di tipo

e interfacce 128–136
input
dalla memoria 494–495
formattazione del,
dalla memoria 495–496
Input/Output 465–584
libreria di 514–518
InputStream 482
leggere da con
FileInputStream 485–486
tipi di 482–483
instanceof dinamico 96–98
intercettazione,
conversione della 226–227
interfacce
aggiunta di 485–486
e informazioni di tipo 128–136
generiche 150–155
parametrizzate, implementazione
delle 230–231
utilizzo delle, per
l'organizzazione 598–605
interfaccia implicita 273
istanze di tipi
creazione di 193–197

## J

JAR (*Java ARchive*) 548
JGA (*Generic Algorithms for
Java*) 284

## L

latent typing 259–262
compensazione alla
mancanza di 265–277
non dannoso per lo strong
typing 263–265
simulazione del, mediante gli
adattatori 273–277
letterali di classe 77–80
utilizzo dei 94–96
libreria di tuple 144–147
limit (limite) 531
limiti 203–208
azione sui 186–191
LinkedHashMap 393–394
List, classe
funzionalità di 368–373
ordinamento e ricerche in 444–446
valutazione di 421–430
lista a collegamento doppio (*dou-
bly linked list*) 373
little endian, approccio 528–529
LRU (*Least Recently
Used*) 388, 393

## M

Map
approfondimenti su 384–393
generatori di 345–349
rendere immutabile una 446–448
valutazione di 434–438
mark (contrassegno) 531
maschere (*template*) 58
meta-annotazioni 645–646
metacaratteri 81, 208–212
di supertipo (*supertype wild-
cards*) 215
senza limiti (*unbounded wild-
card*) 218–226
metodi
applicazione di,
a una sequenza 267–271
generici 155–157
elenchi di 160–161

metodi di classe
estrattore dei 109–112
microbenchmark, rischi 430–431
migration compatibility 183
mixin 250–259
con le interfacce 252–254
con proxy dinamici 257–259
in C++ 250–252
mock, oggetti 128
modelli complessi, costruzione di 174–177
multiple dispatching 631
multithreading 448
MyException, oggetto 10

**N**

nome esteso (*fully qualified name*) 76
null, oggetti 119–128

**O**

oggetto di trasferimento dati (*Data Transfer Object*) 144
operazioni
facoltative 364–368
non supportate 365–368
output su file di testo, variante abbreviata per 497–498
OutputStream
scrivere su un, con FilterOutputStream 487–488
tipi di 484

**P**

parametro di raccolta (*collecting parameter*) 250, 284
perdite di memoria (*memory leak*) 296
persistenza
leggera (*lightweight persistence*) 551
utilizzo della 569–577
position (posizione) 531
prestazioni 387–391
processi, controllo dei 511–514
programmi 2
Pascal 2
proxy dinamici 113–118

**R**

RAD (*Rapid Application Development*) 107
RandomAccessFile, classe 491–492
RandomList 149–150
Reader, classe 488–489
reificazione 183
resumption (ripristino) 7
retrocompatibilità 183
riferimenti, conservazione dei 450–455
riflessione 67, 107–109, 108, 265–267
RMI (*Remote Invocation Method*) 108
RTTI (*RunTime Type Information*) 67
esigenza di 67–70
RuntimeException 30–32

**S**

selezione casuale 597–598
serializzazione di oggetti 551–556
controllo della 557–562
Set
e ordine di archiviazione 373–379

programma di utilità per 166–171
valutazione dei 432–434
sicurezza
dei tipi (*type-safety*) 144
di tipo dinamica 246–247
sintassi di base 643–646
SortedMap 391–393
SortedSet 378–379
sovraccarico 234
specifica di tipo esplicita 159–160
specifiche di eccezione (*exception specification*) 16–17
stack, classe di 148–149
Stack, classe 457–459
standard I/O 508–511
redirezione dello 510–511
standard input 508
leggere da 508–509
standard output 508
static typing 60
Strategy, design pattern 278, 468
Strategy, pattern 325, 328
stub, oggetti 128
System.out
conversione in PrintWriter 509–510

**T**

tag di tipo (*type tag*) 192
Template Method, design pattern 90, 416, 539
test prestazionali 416–421
test unitari (*unit testing*) 668
Throwable, oggetto 5
tipi
autolimitati 236–246
enumerativi 585–640
informazioni sui 67–138
parametrizzati 140
tracciamento dello stack 20–21
transient, parola chiave 563–565
trasmissione multipla (*multiple dispatching*) 609
trusted classes (classi fidate) 71
try, blocco 6
tuple
libreria di 144–147
semplificare l'utilizzo delle 164–166

**U**

@Unit
implementazione di 683–692
utilizzo di, con i generici 681–683
utility 439–444

**V**

values(), metodo 592–596
Vector, classe 456–458
versioni, gestione delle 568
vincoli SQL 657
Visitor, modello, con apt 664–668

**W**

WeakHashMap 453–455
weak key 453
Writer, classe 488–489

**X**

XML 577–581

**Y**

YAGNI (*You Aren't Going to Need It*) 122