

Bruce Eckel

Thinking in Java

I fondamenti

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: 978-8-8719-2303-1
ISBN 10: 88-7192-303-0

Printed in Italy

1st 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	•		
Il polimorfismo	•		
Le interfacce	•		
Le classi interne	•		
Come contenere gli oggetti	•		
Gestione degli errori con le eccezioni		•	
Le stringhe	•		
Informazioni sui tipi		•	
I generici		•	
Gli array		•	
Ancora sui contenitori		•	
Input/Output		•	
I tipi enumerativi		•	
Le 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.

Indice

Prefazione	XVI
Java SE5 e SE6	XVIII
Java SE6	XVIII
La quarta edizione	XIX
Modifiche	XIX
Note sulla grafica di copertina	XXI
Ringraziamenti dell'autore	XXII
Introduzione	XXVI
Prerequisiti	XXVII
Imparare Java	XXVII
Obiettivi	XXVIII
Imparare da questo libro	XXIX
Documentazione JDK in HTML	XXX
Esercizi	XXX
Fondamenti di Java	XXXI
Codice sorgente	XXXI
Standard di codifica	XXXIV
Errori	XXXIV
Capitolo 1 - Introduzione agli oggetti	1
Progresso dell'astrazione	2
Un oggetto possiede un'interfaccia	4
Un oggetto fornisce servizi	7
Implementazione nascosta	8
Riutilizzo dell'implementazione	10
Ereditarietà	11
Relazioni is-a e "is-like-a"	15
Intercambiabilità degli oggetti mediante polimorfismo	17
Gerarchia a radice comune	21
Containeri	22
Tipi parametrizzati o generici	24
Creazione e durata degli oggetti	25
Gestione delle eccezioni e trattamento degli errori	28



Programmazione concorrente	29
Java e Internet	30
Che cos'è il Web?	30
Programmazione client/server	30
Il Web come gigantesco server	31
Programmazione lato client	32
Plug-in	34
Linguaggi di scripting	34
Java	35
Alternative	36
.NET e C#	37
Confronto tra Internet e intranet	37
Programmazione lato server	39
Riepilogo	40
Capitolo 2 - Tutto è un oggetto	41
Gli oggetti si manipolano per riferimento	42
Tutti gli oggetti devono essere creati	43
La registrazione dei dati	43
Un caso speciale: i tipi primitivi	45
Numeri a precisione elevata	46
Gli array in Java	47
Non occorre mai distruggere un oggetto	47
Ambito di visibilità	48
Ambito di visibilità degli oggetti	49
Creazione di nuovi tipi di dato: class	50
Campi e metodi	50
Valori predefiniti per i membri primitivi	52
Metodi, argomenti e valori di ritorno	53
Elenco degli argomenti	54
Come costruire un programma Java	55
Visibilità del nome	55
Utilizzo di altri componenti	56
La parola chiave static	57
Il vostro primo programma Java	60
Compilazione ed esecuzione	62
Commenti e documentazione incorporata	63
Commento di documentazione	64
Sintassi	65
Inclusione diretta di codice HTML	66
Alcuni tag di esempio	67
Esempio di documentazione	70
Stile di codifica	71
Riepilogo	72
Esercizi	72

Capitolo 3 - Gli operatori	75
La dichiarazione più semplice: print	75
Utilizzo degli operatori Java	77
Precedenza	77
Assegnazione	78
Aliasing in fase di chiamata di metodo	80
Operatori matematici	81
Operatori unari di segno positivo e negativo	84
Incremento e decremento automatici	85
Operatori relazionali	86
Test sull'equivalenza degli oggetti	87
Operatori logici	89
Cortocircuito	91
Valori letterali	92
Notazione esponenziale	94
Gli operatori bit a bit (bitwise)	96
Gli operatori di tipo shift	97
Operatore ternario: if-else	102
Operatori di String + e +=	104
Problemi ricorrenti nell'utilizzo degli operatori	105
Operatori di cast	106
Troncamento e arrotondamento	107
Promozione	109
Java non ha "sizeof"	109
Riepilogo degli operatori	109
Riepilogo	122
Capitolo 4 - Il controllo dell'esecuzione	123
true e false	123
if-else	124
Iterazione	125
do-while	126
for	127
L'operatore virgola	129
La sintassi foreach	130
return	133
break e continue	134
L'"abominevole" goto	136
switch	141
Riepilogo	145
Capitolo 5 - Inizializzazione e cleanup	147
L'inizializzazione è garantita dal costruttore	148
Overloading dei metodi	150



Come distinguere i metodi overloaded	153	Inizializzazione della superclasse	247
Overloading con primitivi	154	Costruttori con argomenti	248
Overloading sui valori di ritorno	159	Delega	250
Costruttori predefiniti	160	Combinazione di composizione ed ereditarietà	252
Parola chiave this	162	Come garantire un cleanup corretto	255
Chiamate di costruttori da costruttori	165	Occultamento del nome	259
Significato di static	167	Come scegliere tra composizione ed ereditarietà	261
Cleanup: finalizzazione e garbage collection	168	L'accesso protected	264
A che cosa serve finalize()?	169	Upcasting	265
È necessario fare pulizia	170	Perché "upcasting"?	267
Condizione di terminazione	171	Nuovo confronto tra composizione ed ereditarietà	267
Funzionamento della garbage collection	173	La parola chiave final	268
Inizializzazione dei membri	177	Dati di tipo final	268
Come specificare l'inizializzazione	179	Campi final non inizializzati	271
Inizializzazione del costruttore	181	Gli argomenti final	273
Ordine di inizializzazione	182	Metodi di tipo final	274
Inizializzazione di dati static	183	final e private	275
Inizializzazione static esplicita	187	Classi di tipo final	277
Inizializzazione di istanze non static	189	Importanti considerazioni su final	278
Inizializzazione di array	191	Inizializzazione e caricamento delle classi	279
Elenchi di argomenti variabili	197	Inizializzazione con ereditarietà	280
Tipi enumerativi	204	Riepilogo	282
Riepilogo	207	Capitolo 8 - Polimorfismo	285
Capitolo 6 - Controllo di accesso	209	Revisione dell'upcasting	286
Package: l'unità di libreria	211	Dimenticate il tipo di oggetto	288
Organizzazione del codice	212	La svolta	290
Come creare nomi di package univoci	214	Binding delle chiamate a metodi	290
Collisioni	218	Come ottenere il comportamento corretto	291
Librerie personalizzate di strumenti	219	Estensibilità	296
Utilizzo di import per modificare i comportamenti	222	Attenzione alla sovrascrittura dei metodi private!	300
Osservazioni sui package	222	Attenzione ai metodi e ai campi static!	300
Modificatori di accesso in Java	223	Costruttori e polimorfismo	303
Package access	223	Ordine delle chiamate al costruttore	303
public: accesso di interfaccia	224	Ereditarietà e cleanup	306
Package predefinito	225	Comportamento dei metodi polimorfi all'interno dei costruttori	312
private: proibito toccare!	226	Tipi di ritorno covarianti	315
protected: accesso per ereditarietà	228	Progettazione in funzione dell'ereditarietà	316
Interfaccia e implementazione	231	Confronto tra sostituzione ed estensione	318
Accesso alle classi	232	Downcasting e tipi di informazioni a runtime	321
Riepilogo	237	Riepilogo	323
Capitolo 7 - Riutilizzo delle classi	239	Capitolo 9 - Le interfacce	325
Sintassi della composizione	240	Metodi e classi astratti	325
Sintassi dell'ereditarietà	244	Interfacce	331



Dissociazione completa	336
"Ereditarietà multipla" in Java	343
Estensione di un'interfaccia mediante ereditarietà	346
Collisione di nomi nella combinazione di interfacce	348
Come adattarsi a un'interfaccia	349
Campi nelle interfacce	352
Inizializzare i campi nelle interfacce	353
Interfacce nidificate	354
Interfacce e factory	358
Riepilogo	362
Capitolo 10 - Le classi interne	363

Creazione delle classi interne	364
Collegamento alla classe esterna	366
Utilizzo di .this e .new	369
Classi interne e upcasting	371
Classi interne, metodi e ambiti	374
Classi interne anonime	377
Nuove considerazioni su Factory Method	383
Classi nidificate	386
Classi all'interno di interfacce	388
Accesso dall'interno di classi a nidificazione multipla	390
Perché utilizzare le classi interne?	391
Closure e callback	394
Classi interne e framework di controllo	397
Come ereditare dalle classi interne	406
È possibile sovrascrivere le classi interne?	407
Classi interne locali	409
Identificatori delle classi interne	412
Riepilogo	413
Capitolo 11 - Come contenere gli oggetti	415

Confronto tra Collection e Iterator	458
Foreach e iteratori	462
L'Adapter Method idiom	466
Riepilogo	470

Capitolo 12 - Stringhe **475**

Stringhe invariabili	475
Confronto tra sovraccarico di "+" e StringBuilder	477
Ricorrenza non intenzionale	483
Operazioni sulle stringhe	485
Formattazione dell'output	487
printf()	488
System.out.format()	488
Classe Formatter	489
Modificatori di formato	490
Conversioni di Formatter	492
String.format()	496
Uno strumento per il dump esadecimale	497
Espressioni regolari	498
Principi fondamentali	499
Creazione di espressioni regolari	503
Quantificatori	505
CharSequence	506
Pattern e Matcher	507
Gruppi	511
Flag dei modelli	515
split()	518
Operazioni di sostituzione	519
reset()	522
Espressioni regolari e Java I/O	522
Scansione dell'input	525
Separatori di Scanner	528
Scansione con le espressioni regolari	529
StringTokenizer	530
Riepilogo	531

Appendice A - Supplementi **533**

Supplementi scaricabili	533
Thinking in C: i fondamenti di Java	534
Seminari Thinking in Java	534
Seminario Hands-On Java su CD	534
Seminario Thinking in Objects	535
Thinking in Enterprise Java	535
Thinking in Patterns (with Java)	536
Seminario Thinking in Patterns	536
Consulenza e revisione di progetti	537

**Appendice B - Risorse****539**

Software	539
Editor e ambienti IDE	540
Libri	540
Analisi e progettazione	541
Python	544
Bibliografia dell'autore	545

Indice analitico**547**

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 programmati 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 programmati 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 programmati 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 accumulate 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.

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

Introduzione agli oggetti

"Se dissezioniamo la natura, se la organizziamo per concetti e le attribuiamo significati come facciamo, è perché siamo parti di un comune accordo [...] valido per tutta la comunità linguistica, che è codificato nelle strutture della nostra lingua. [...] per essere in grado di parlare dobbiamo appartenere all'organizzazione, e aderire alla classificazione dei dati stabilita da questo accordo."

*Benjamin Lee Whorf (1897-1941),
Linguaggio, pensiero e realtà, Torino, Boringhieri, 1970.*



L'origine della rivoluzione informatica è in una macchina, pertanto la genesi dei nostri linguaggi di programmazione tende ad assomigliare a questa macchina.

Ma i computer non sono tanto *macchine*, quanto *strumenti* di amplificazione del pensiero, un mezzo espressivo di tipo diverso: "biciclette per la mente", come ama dire Steve Jobs. Di conseguenza gli strumenti stanno iniziando ad assomigliare meno alle macchine e più a parti delle nostre menti, nonché ad altre forme di espressione, quali la scrittura, la pittura, la scultura, l'animazione e il cinema. La programmazione orientata agli oggetti (OOP, *Object-Oriented Programming*) fa parte di questo movimento, orientato all'utilizzo del computer come mezzo espressivo.

Il capitolo introduce i concetti fondamentali dell'OOP, e include una descrizione dei metodi di sviluppo. Questo capitolo, e il manuale in genere, presuppongono una certa esperienza di programmazione, anche se non necessariamente in linguaggio C. Se ritenete di avere bisogno di maggiore preparazione nella programmazione prima di affrontare questo manuale, potrete usufruire del seminario multimediale *Thinking in C*, scaricabile in lingua inglese dal sito www.mindview.net.

Questo capitolo è integrativo e va considerato come un supporto, dal momento che molti lettori hanno perplessità ad affrontare la programmazione orientata



agli oggetti senza averne prima compreso i concetti fondamentali: verranno presentate le nozioni necessarie per una valida introduzione all'OOP. D'altro canto altri lettori non riescono a cogliere il concetto nella sua interezza fino a quando non hanno visto i componenti all'opera; si tratta di persone che arrivano anche a sentirsi perse se non hanno codice su cui "mettere le mani". Se appartenete a questa categoria e siete ansiosi di affrontare le specifiche del linguaggio, non esitate a saltare il capitolo; ciò non vi impedirà di scrivere programmi né di imparare il linguaggio. All'occorrenza, in qualsiasi momento potrete rivedere questo capitolo per completare le vostre competenze e comprendere l'importanza degli oggetti e della loro progettazione.

Progresso dell'astrazione

Tutti i linguaggi di programmazione forniscono un certo livello di astrazione. Se ne può dedurre che la complessità dei problemi risolvibili è direttamente correlata al *genere* e alla *qualità* di astrazione fornita. La parola "genere" risponde alla domanda: "che cosa si sta astraendo?". Il linguaggio Assembler, per esempio, fornisce una limitata astrazione del sistema di base. Molti dei linguaggi cosiddetti "imperativi" che sono seguiti, quali FORTRAN, BASIC e C, erano astrazioni del linguaggio Assembler.

Questi linguaggi rappresentano un notevole miglioramento rispetto all'Assembler, tuttavia richiedono di pensare in termini di struttura del computer, anziché di struttura del problema da risolvere. Il programmatore deve stabilire un'associazione tra il "modello di macchina" (sistema), nello *spazio della soluzione*, vale a dire il "luogo" dove implementare la soluzione (un computer, per esempio), e il modello del problema che viene effettivamente risolto, il cosiddetto *spazio del problema*, ossia il luogo in cui è presente il problema (un'azienda, per esempio). Lo sforzo richiesto per eseguire questa corrispondenza e il fatto che essa è esterna al linguaggio di programmazione hanno prodotto programmi difficili da scrivere e di manutenzione onerosa, e come effetto collaterale hanno dato origine all'intera industria dei "metodi di programmazione".

L'alternativa alla *modellizzazione* della macchina consiste nel *configurare* il problema da risolvere: i primi linguaggi di programmazione, come LISP e APL, hanno optato per una particolare visione della realtà, rispettivamente con "tutti i problemi sono elenchi" e "tutti i problemi sono algoritmici". Il Prolog riduce tutti i problemi a catene di decisioni. I linguaggi sono stati ideati per la programmazione basata sui vincoli o tramite la manipolazione di simboli grafici; quest'ultima tecnica, tuttavia, si è dimostrata eccessivamente restrittiva. Entrambi questi approcci costituiscono una valida solu-



zione alla particolare classe di problema per la quale sono stati progettati, ma quando si esula da questo ambito, questi linguaggi diventano perlomeno difficili da usare.

L'approccio alla programmazione orientata agli oggetti si spinge oltre, fornendo strumenti per rappresentare gli elementi nello spazio del problema, secondo una rappresentazione abbastanza generica da non limitare il programmatore a una specifica classe di problema. Ci si riferisce agli elementi nello spazio del problema e alle loro rappresentazioni nello spazio della soluzione con il termine *oggetti* (tenete presente che sono necessari anche altri oggetti che non trovano corrispondenza nello spazio del problema). Aggiungendo nuovi tipi di oggetti il programma è in grado di adattarsi al problema particolare; in questo modo, leggendo il codice che descrive la soluzione si legge anche la descrizione del problema.

Questa è l'astrazione del linguaggio più flessibile e potente che si sia mai verificata.¹

Mediante questo approccio la OOP permette di descrivere il problema nei termini del problema stesso, invece che in quelli del sistema sul quale la soluzione deve essere eseguita. Esiste in ogni caso un collegamento al computer: ogni oggetto ricorda in qualche modo un piccolo computer, poiché possiede uno stato e operazioni che gli si può ordinare di eseguire. Comunque questa analogia può essere valida anche per gli oggetti del mondo reale: tutti gli oggetti possiedono caratteristiche e comportamenti.

Alan Kay ha riassunto le cinque caratteristiche fondamentali di Smalltalk, il primo linguaggio OOP di successo, dal quale è parzialmente derivato Java. Queste caratteristiche evidenziano un approccio puro alla programmazione a oggetti.

- 1. Ogni cosa è un oggetto.** Considerate un oggetto come una variabile elaborata che conserva dati, ma alla quale è anche possibile inviare "richieste", ordinando l'esecuzione di operazioni su se stessa. In teoria è possibile prendere qualsiasi componente concettuale del problema da risolvere, cani, edifici, servizi ecc. e rappresentarlo come oggetto nel programma.
- 2. Un programma è un insieme di oggetti che si ordinano reciprocamente che cosa fare, scambiandosi messaggi.** Per eseguire una richiesta a un oggetto occorre "inviargli un messaggio": in termini più concreti, si può pensare

¹ Alcuni progettisti di linguaggi hanno deciso che la programmazione a oggetti da sola non sia adatta per risolvere in modo semplice tutti i problemi di programmazione, sostenendo la combinazione di orientamenti diversi in linguaggi di programmazione a paradigmi multipli. Si veda in proposito *Multiparadigm Programming in Leda* di Timothy Budd (Addison-Wesley, 1995).



agli oggetti senza averne prima compreso i concetti fondamentali: verranno presentate le nozioni necessarie per una valida introduzione all'OOP. D'altro canto altri lettori non riescono a cogliere il concetto nella sua interezza fino a quando non hanno visto i componenti all'opera; si tratta di persone che arrivano anche a sentirsi perse se non hanno codice su cui "mettere le mani". Se appartenete a questa categoria e siete ansiosi di affrontare le specifiche del linguaggio, non esitate a saltare il capitolo; ciò non vi impedirà di scrivere programmi né di imparare il linguaggio. All'occorrenza, in qualsiasi momento potrete rivedere questo capitolo per completare le vostre competenze e comprendere l'importanza degli oggetti e della loro progettazione.

Progresso dell'astrazione

Tutti i linguaggi di programmazione forniscono un certo livello di astrazione. Se ne può dedurre che la complessità dei problemi risolvibili è direttamente correlata al *genere* e alla *qualità* di astrazione fornita. La parola "genere" risponde alla domanda: "che cosa si sta astraendo?". Il linguaggio Assembler, per esempio, fornisce una limitata astrazione del sistema di base. Molti dei linguaggi cosiddetti "imperativi" che sono seguiti, quali FORTRAN, BASIC e C, erano astrazioni del linguaggio Assembler.

Questi linguaggi rappresentano un notevole miglioramento rispetto all'Assembler, tuttavia richiedono di pensare in termini di struttura del computer, anziché di struttura del problema da risolvere. Il programmatore deve stabilire un'associazione tra il "modello di macchina" (sistema), nello *spazio della soluzione*, vale a dire il "luogo" dove implementare la soluzione (un computer, per esempio), e il modello del problema che viene effettivamente risolto, il cosiddetto *spazio del problema*, ossia il luogo in cui è presente il problema (un'azienda, per esempio). Lo sforzo richiesto per eseguire questa corrispondenza e il fatto che essa è esterna al linguaggio di programmazione hanno prodotto programmi difficili da scrivere e di manutenzione onerosa, e come effetto collaterale hanno dato origine all'intera industria dei "metodi di programmazione".

L'alternativa alla *modellizzazione* della macchina consiste nel *configurare* il problema da risolvere: i primi linguaggi di programmazione, come LISP e APL, hanno optato per una particolare visione della realtà, rispettivamente con "tutti i problemi sono elenchi" e "tutti i problemi sono algoritmici". Il Prolog riduce tutti i problemi a catene di decisioni. I linguaggi sono stati ideati per la programmazione basata sui vincoli o tramite la manipolazione di simboli grafici; quest'ultima tecnica, tuttavia, si è dimostrata eccessivamente restrittiva. Entrambi questi approcci costituiscono una valida solu-



zione alla particolare classe di problema per la quale sono stati progettati, ma quando si esula da questo ambito, questi linguaggi diventano perlomeno difficili da usare.

L'approccio alla programmazione orientata agli oggetti si spinge oltre, fornendo strumenti per rappresentare gli elementi nello spazio del problema, secondo una rappresentazione abbastanza generica da non limitare il programmatore a una specifica classe di problema. Ci si riferisce agli elementi nello spazio del problema e alle loro rappresentazioni nello spazio della soluzione con il termine *oggetti* (tenete presente che sono necessari anche altri oggetti che non trovano corrispondenza nello spazio del problema). Aggiungendo nuovi tipi di oggetti il programma è in grado di adattarsi al problema particolare; in questo modo, leggendo il codice che descrive la soluzione si legge anche la descrizione del problema.

Questa è l'astrazione del linguaggio più flessibile e potente che si sia mai verificata.¹

Mediante questo approccio la OOP permette di descrivere il problema nei termini del problema stesso, invece che in quelli del sistema sul quale la soluzione deve essere eseguita. Esiste in ogni caso un collegamento al computer: ogni oggetto ricorda in qualche modo un piccolo computer, poiché possiede uno stato e operazioni che gli si può ordinare di eseguire. Comunque questa analogia può essere valida anche per gli oggetti del mondo reale: tutti gli oggetti possiedono caratteristiche e comportamenti.

Alan Kay ha riassunto le cinque caratteristiche fondamentali di Smalltalk, il primo linguaggio OOP di successo, dal quale è parzialmente derivato Java. Queste caratteristiche evidenziano un approccio puro alla programmazione a oggetti.

- Ogni cosa è un oggetto.** Considerate un oggetto come una variabile elaborata che conserva dati, ma alla quale è anche possibile inviare "richieste", ordinando l'esecuzione di operazioni su se stessa. In teoria è possibile prendere qualsiasi componente concettuale del problema da risolvere, cani, edifici, servizi ecc. e rappresentarlo come oggetto nel programma.
- Un programma è un insieme di oggetti che si ordinano reciprocamente che cosa fare, scambiandosi messaggi.** Per eseguire una richiesta a un oggetto occorre "inviergli un messaggio": in termini più concreti, si può pensare

1. Alcuni progettisti di linguaggi hanno deciso che la programmazione a oggetti da sola non sia adatta per risolvere in modo semplice tutti i problemi di programmazione, sostenendo la combinazione di orientamenti diversi in linguaggi di programmazione a paradigmi multipli. Si veda in proposito *Multiparadigm Programming in Leda* di Timothy Budd (Addison-Wesley, 1995).



a un messaggio come alla chiamata a un metodo che appartiene a un particolare oggetto.

3. **Ogni oggetto possiede una propria memoria, composta da altri oggetti.** Da un punto di vista diverso, significa che è possibile creare un nuovo genere di oggetto assemblando oggetti esistenti.² In questo modo è possibile realizzare programmi complessi, pur celandoli dietro la semplicità dei singoli oggetti.
4. **Ogni oggetto possiede un tipo.** In gergo informatico ogni oggetto è un'istanza di una classe, in cui "classe" è sinonimo di "tipo". La principale caratteristica distintiva di una classe è il tipo di messaggi che è possibile inviarle.
5. **Tutti gli oggetti di uno stesso tipo possono ricevere gli stessi messaggi.** Si tratta, in effetti, di una sorta di "dichiarazione caricata" (*loaded statement*), come vedrete in seguito. Poiché un oggetto di tipo "cerchio" è anche un oggetto di tipo "forma", un cerchio è sempre in grado di accettare i messaggi destinati a una forma. Questo significa che potete scrivere codice per comunicare con oggetti di tipo forma, e trattare automaticamente qualsiasi cosa si adatti alla descrizione di una forma. Questa sostituibilità è uno dei concetti più potenti dell'OOP.

Booch fornisce una descrizione ancora più sintetica: *Un oggetto possiede uno stato, un comportamento e un'identità*.

Ciò significa che l'oggetto può avere dati interni, che gli conferiscono uno stato, metodi, per produrre il comportamento, e che ciascun oggetto può essere distinto in modo univoco da ogni altro: più concretamente, ogni oggetto in memoria ha un indirizzo univoco.²

Un oggetto possiede un'interfaccia

Si ritiene che Aristotele sia stato il primo ad analizzare il concetto di tipo: egli parlò della "classe dei pesci e della classe degli uccelli". L'idea che tutti gli oggetti, pur essendo unici, siano anche parte di una classe di oggetti che condividono caratteristiche e comportamenti fu usata direttamente nel primo linguaggio orientato agli oggetti, il Simula-67, con la sua parola chiave fondamentale, **class**, che introduce un nuovo tipo in un programma.

2. In realtà questa definizione è piuttosto restrittiva, tenuto conto che teoricamente gli oggetti possono esistere su computer e spazi di memoria diversi, nonché essere registrati su disco. In questi casi l'identità dell'oggetto deve essere determinata con criteri diversi dall'indirizzamento in memoria.



Simula, come indica il nome, è stato creato per lo sviluppo di simulazioni, come il classico "problema del cassiere di banca" nel quale vi sono numerosi "oggetti", rappresentati da cassieri, clienti, conti, transazioni e unità di denaro. Si tratta di oggetti identici tra loro che, fatta eccezione per il loro stato durante l'esecuzione di un programma, vengono raggruppati in "classi di oggetti": da questa raffigurazione proviene la parola chiave **class**. La creazione di tipi di dato astratti (classi) è un concetto fondamentale della programmazione a oggetti. I tipi di dato astratti operano quasi esattamente come i tipi di dato nativi: consentono di creare variabili di un determinato tipo, chiamate in gergo *oggetti* o *istanze*, e di manipolarle, inviando *messaggi* o *richieste*. È sufficiente inviare un messaggio e l'oggetto esegue quanto gli è stato ordinato di fare.

I membri (elementi) di ogni classe condividono alcune caratteristiche: ogni conto ha un saldo, ogni cassiere può accettare un deposito e così via. Analogamente ogni membro ha il proprio stato: ogni conto ha un saldo differente, ogni cassiere un nome diverso. In questo modo cassieri, clienti, conti, transazioni ecc. possono essere rappresentati come singole entità univoche nel programma. Questa entità è l'oggetto; ogni oggetto appartiene a una particolare classe che ne definisce caratteristiche e comportamenti.

Così, benché effettivamente nella programmazione a oggetti si creino nuovi tipi di dato, in teoria tutti i linguaggi di programmazione a oggetti fanno uso della parola chiave **class**. Quando si riscontra la parola chiave **type** è sufficiente pensare a **class**, e viceversa.³

Poiché la classe descrive un insieme di oggetti con caratteristiche (dati) e comportamenti (funzionalità) identici, essa è un vero e proprio tipo di dato, in quanto un numero in virgola mobile, per esempio, possiede anche un insieme di caratteristiche e comportamenti. La differenza consiste nel fatto che il programmatore definisce una classe in modo tale da adattarla al problema, invece di essere costretto a utilizzare un tipo di dato esistente che è stato progettato per rappresentare un'unità di archiviazione in un computer. È quindi possibile estendere il linguaggio di programmazione aggiungendo nuovi tipi di dato specifici per le necessità contingenti. Il sistema di programmazione non ha problemi ad accettare le nuove classi e fornisce loro le caratteristiche e la verifica di tipo di cui beneficiano i tipi di dato nativi.

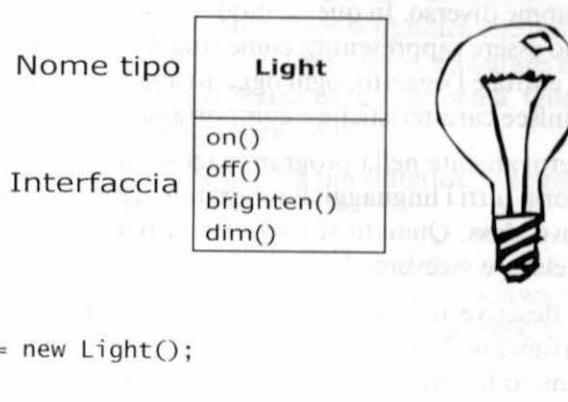
L'approccio orientato agli oggetti non si limita alla costruzione di simulazioni: che accettiate o meno il concetto che ogni programma è una simulazione del sistema che state progettando, il ricorso alle tecniche OOP può ridurre

3. Alcuni programmatori distinguono i due termini, sostenendo che il *tipo* determina l'interfaccia, la cui implementazione specifica è rappresentata dalla *classe*.



facilmente un grande insieme di problemi a una soluzione semplice. Dopo aver impostato una classe è possibile creare qualsiasi numero di oggetti di quella classe si ritenga opportuno, poi manipolare tali oggetti come se fossero gli elementi presenti nel problema da risolvere. In effetti, una delle sfide della programmazione a oggetti consiste nel realizzare una corrispondenza di tipo *uno-a-uno* tra gli elementi nello spazio del problema e gli oggetti nello spazio della soluzione.

Come fare, tuttavia, per creare un oggetto utile e funzionale? A questo proposito è essenziale poter inviare una richiesta all'oggetto in modo che esegua un'operazione, per esempio completare una transazione, disegnare qualcosa sullo schermo o accendere un interruttore. Inoltre, ogni oggetto può soddisfare solo determinate richieste. Le richieste inoltrabili a un oggetto sono definite dalla sua interfaccia, e il tipo è ciò che determina l'interfaccia. Un semplice esempio potrebbe essere la rappresentazione di una lampadina.



L'interfaccia determina le richieste che è possibile eseguire su un particolare oggetto. Tuttavia in qualche punto deve esserci il codice che permette di soddisfare la richiesta: questo, insieme ai dati encapsulati, costituisce l'implementazione. Dal punto di vista della programmazione procedurale il concetto non è così complesso. A un tipo è associato un metodo per ogni possibile richiesta; quando si esegue una determinata richiesta all'oggetto viene richiamato il relativo metodo. Questo processo è generalmente riassunto dall'espressione "invia un messaggio a un oggetto", vale a dire eseguire una richiesta: gli oggetti si comporteranno conformemente a questo messaggio, ossia "eseguiranno il codice".

In questo esempio il nome del tipo/classe è **Light**, il nome dello specifico oggetto-lampadina è **lt** e le richieste inviabili all'oggetto prevedono rispettivamente di accenderlo, spegnerlo, renderlo più brillante o meno luminoso.



Si crea un oggetto **Light** definendo per esso un "riferimento" (**lt**) e chiamando la parola chiave **new** per richiedere un nuovo oggetto di questo tipo. Per inviargli un messaggio basta dichiarare il nome dell'oggetto, collegandolo tramite un punto (.) alla richiesta di messaggio. Dal punto di vista dell'utente di una classe predefinita questo è quasi tutto ciò che occorre conoscere per programmare con gli oggetti.

Il diagramma precedente è conforme al formato definito dal linguaggio UML (*Unified Modeling Language*). Ogni classe è rappresentata da un rettangolo, nella parte superiore del quale è riportato il *nome del tipo*, nella parte centrale è presente qualsiasi *membro dati* che riteniate di descrivere e nella parte inferiore sono indicati i *metodi*, ossia le funzioni che appartengono all'oggetto e che ricevono i messaggi inviati all'oggetto stesso. Spesso i diagrammi di progettazione UML mostrano soltanto il nome della classe e i metodi pubblici, cosicché la parte centrale è assente, come in questo caso; ugualmente, se ritenete di indicare soltanto il nome della classe la parte inferiore non dovrà essere mostrata.

Un oggetto fornisce servizi

Quando dovete definire o comprendere la progettazione di un programma, uno degli approcci migliori è considerare gli oggetti come "fornitori di servizi": sarà il vostro programma a fornire i servizi agli utenti, ricorrendo a sua volta ai servizi messi a disposizione da altri oggetti. Il vostro compito consiste nel produrre (o, ancora meglio, individuare nelle librerie di codice esistenti) un insieme di oggetti capaci di fornire i servizi adeguati per risolvere il problema.

Un modo per procedere in questo senso è chiedersi: "se magicamente potessi estrarli da un cappello, quali oggetti mi servirebbero adesso?". Per esempio, supponete di realizzare un programma di contabilità bancaria. Vi occorreranno sicuramente oggetti contenenti le schermate di inserimento predefinite, un altro insieme di oggetti che esegua calcoli contabili, e un oggetto che gestisca la stampa di assegni e fatture sui diversi tipi di stampante.

Alcuni di questi oggetti potrebbero essere già presenti, ma quelli che non esistono a che cosa dovrebbero assomigliare? Quali servizi dovrebbero fornire questi oggetti e di quali altri oggetti avrebbero bisogno per adempiere i loro compiti? Abituandovi a porvi queste domande riuscirete via via ad affermare "questo oggetto sembra abbastanza semplice da poter essere scritto", oppure "sono certo che questo oggetto esiste già". Si tratta di un approccio ragionevole per scomporre un problema in un insieme di oggetti.



Pensare a un oggetto in termini di fornitore di servizi comporta il beneficio aggiuntivo di contribuire al miglioramento della coesione dell'oggetto. Un'elevata coesione è qualità essenziale della progettazione software: significa che i diversi aspetti di un componente software, per esempio un oggetto (benché ciò possa applicarsi anche a un metodo o a una libreria di oggetti), "stanno bene insieme". Tuttavia, quando si cerca di progettare oggetti dotati di eccessive funzionalità spesso si verificano problemi.

Per esempio, nel modulo di controllo della stampa potreste ritenere di avere bisogno di un oggetto che conosca tutto sulla formattazione e la stampa. Probabilmente scoprirete che questo è troppo per un oggetto singolo e che risulta più pratico disporre di tre o più oggetti. Uno di essi potrebbe rappresentare un catalogo di tutti i possibili modelli di assegni, che possa essere interrogato sulle informazioni per stampare un particolare modello. Un altro oggetto, o insieme di oggetti, potrebbe essere un'interfaccia di stampa generica che conosca tutti i vari tipi di stampanti, ma niente di relativo alla contabilità: questo è il tipico oggetto da acquistare già pronto, piuttosto che scrivere da soli. Un terzo oggetto, infine, potrebbe utilizzare i servizi offerti dagli altri due per eseguire la stampa. Così facendo ogni oggetto avrà un insieme coesivo (coerente) di servizi offerti. In una buona progettazione OOP ogni oggetto è adatto a un compito, lo esegue nel migliore dei modi, ma evita di offrire troppe funzionalità.

Questo modo di procedere vi permette di determinare quali oggetti potrebbero essere acquistati (l'interfaccia alle stampanti), ma anche di scoprire nuovi oggetti che potrebbero essere riutilizzati altrove (il catalogo dei modelli di assegni).

Considerare gli oggetti come fornitori di servizi rappresenta una grande semplificazione, utile in fase di progettazione ma anche quando altri programmati devono interpretare il codice esistente o capire come riutilizzare un oggetto: se riescono a vedere il valore dell'oggetto in funzione del servizio fornito è molto più facile che possano servirsene nuovamente nel progetto.

Implementazione nascosta

È utile suddividere il terreno di gioco in *creatori di classi*, ossia quelli che generano nuovi tipi di dato, e *programmati client*, vale a dire gli "utilizzatori" delle classi, che si servono dei tipi di dato nelle proprie applicazioni.⁴

4. L'autore ringrazia il suo amico Scott Meyers per aver suggerito il termine *programmati client* (*client programmer*).



L'obiettivo del programmatore client è raccogliere un insieme di classi da utilizzare nello sviluppo rapido di applicazioni; lo scopo del creatore di classi, invece, è costruire una classe che espone soltanto quanto è necessario al programmatore client, nascondendo tutto il resto. La ragione di questo è da ricercarsi nel fatto che, se l'implementazione è nascosta, il programmatore client non può accedervi, e il creatore di classi può cambiare la parte nascosta come ritiene opportuno, senza preoccuparsi dell'eventuale impatto su altri "utilizzatori".

Di norma la porzione nascosta rappresenta la parte interna di un oggetto che potrebbe essere facilmente alterata da un programmatore client imprudente o poco informato, cosicché celando l'implementazione si riduce il rischio di bug nel programma.

In qualsiasi rapporto è importante definire limiti che siano rispettati da tutte le parti coinvolte. Quando create una libreria, stabilite un rapporto con il programmatore client, che è anch'egli uno sviluppatore, ma si occupa di costruire l'applicazione utilizzando le librerie fornite, magari per realizzarne una più grande.

Se tutti i membri di una classe fossero disponibili a chiunque, allora il programmatore client potrebbe fare qualsiasi cosa con tale classe, e non vi sarebbe modo di imporre alcuna regola. Benché sia auspicabile che il programmatore client non possa manipolare in modo diretto membri della vostra classe, senza controllo di accesso non vi sarà possibile evitarlo.

Pertanto, il primo motivo che giustifica l'esistenza del controllo di accesso è tenere i programmati client lontani dai componenti di un programma che dovrebbero essere inaccessibili: porzioni indispensabili per l'operatività interna del tipo di dato, che tuttavia non fanno parte dell'interfaccia richiesta dagli utenti per risolvere i loro specifici problemi. In effetti questo *modus operandi* è utile ai programmati client, che possono così distinguere facilmente ciò che è importante per loro da quanto invece sono liberi di ignorare.

Il secondo motivo per controllare l'accesso è consentire al progettista delle librerie di modificare i meccanismi interni di una classe senza preoccuparsi degli effetti che i cambiamenti potrebbero avere sul programmatore client. Per esempio potreste realizzare una particolare classe in modo semplice per facilitarne lo sviluppo, e più tardi scoprire che è necessario riscriverla per migliorarne la velocità di esecuzione. Se l'interfaccia e l'implementazione sono nettamente separate e protette, questa operazione sarà molto semplice.

Java utilizza tre parole chiave esplicite per imporre limiti in una classe: **public**, **private** e **protected**. Questi "specificatori d'accesso" determinano chi può utilizzare le definizioni da esse precedute. La parola chiave **public** indica

che l'elemento che la segue è disponibile per chiunque; **private** segnala che nessuno può accedere all'elemento tranne il creatore del tipo, all'interno dei metodi di quel tipo: **private** erige un muro insormontabile tra voi e il programmatore client, che provando ad accedere a un membro privato otterrà un errore di compilazione. Infine, la parola chiave **protected** agisce come **private**, con l'eccezione che la classe che eredita ha accesso ai membri **protected**, non a quelli **private**. Il concetto di ereditarietà sarà introdotto tra breve.

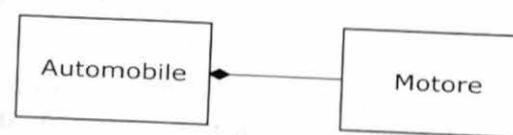
Java offre anche un accesso predefinito, che entra in gioco quando non si utilizza uno dei suddetti specificatori. Questo è di solito definito come *package access*, poiché le classi possono accedere ai membri di altre classi nello stesso componente di libreria, mentre al di fuori del package gli stessi membri appaiono come **private**.

Riutilizzo dell'implementazione

Quando una classe è stata creata e testata dovrebbe rappresentare, almeno in teoria, un'unità di codice utile. Si dà il caso, però, che questa "riutilizzabilità" non sia così facile da perseguire quanto si potrebbe sperare; la progettazione di un oggetto riutilizzabile richiede esperienza e intuizione. Tuttavia, quando si è definita questa progettazione l'oggetto non aspetta altro che di essere riutilizzato. La caratteristica di riutilizzo del codice è uno dei principali vantaggi forniti dai linguaggi di programmazione a oggetti.

Il modo più semplice per riutilizzare una classe consiste nell'impiegare direttamente un suo oggetto; è comunque possibile anche inserire un oggetto di questa classe in una nuova, ciò che in gergo si definisce "creare un membro oggetto". La nuova classe può essere composta da qualsiasi numero e tipo di altri oggetti, in qualsiasi combinazione necessaria per ottenere le funzionalità richieste dalla nuova classe.

Poiché si tratta di comporre una nuova classe assemblando classi esistenti, questo concetto è chiamato *composizione*; se la composizione avviene dinamicamente, di solito è chiamata *aggregazione*. La composizione è spesso citata con un rapporto di tipo "possiede-un", come nello schema seguente, nel quale "un'automobile ha un motore".



Questo diagramma UML indica la composizione con un piccolo rombo di colore nero pieno, che dichiara l'esistenza di un oggetto di tipo **Automobile**. L'autore di solito si serve di una forma più semplice: una normale linea, senza il rombo, per indicare un'associazione.⁵

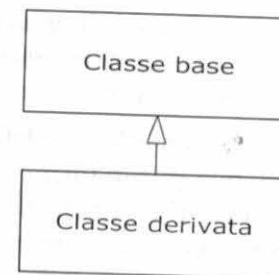
La composizione consente una flessibilità considerevole. In genere gli oggetti-membro della nuova classe sono privati (dichiarati come **private**), il che li rende inaccessibili ai programmatore client che usano la classe. Questo permette di modificare tali membri senza interferire sul codice client esistente. È anche possibile modificare gli oggetti-membro in fase di esecuzione per variare dinamicamente il comportamento del programma. L'ereditarietà, descritta di seguito, non offre questa flessibilità: in fase di compilazione, infatti, il compilatore deve imporre restrizioni sulle classi create mediante ereditarietà.

Dal momento che nella programmazione a oggetti l'ereditarietà è così importante, viene spesso enfatizzata oltremisura, di conseguenza il neoprogrammatore può riceverne l'impressione che debba essere usata ovunque: un preconcetto che può portare a progetti poco eleganti ed eccessivamente complessi. Al contrario, durante la creazione di nuove classi è necessario valutare per prima cosa la composizione, poiché più semplice e flessibile. Ricorrendo a questo approccio i progetti risulteranno più lineari e puliti. Dopo aver acquisito sufficiente esperienza, vi apparirà evidente quando utilizzare l'ereditarietà.

Ereditarietà

In sé, l'idea di utilizzare un oggetto è utile perché permette di raggruppare dati e funzionalità per concetti, in modo da rappresentare un'idea appropriata nello spazio del problema, invece di essere costretti a usare il linguaggio del computer sottostante. Questi concetti sono espressi come unità fondamentali nel linguaggio di programmazione mediante la parola chiave **class**. Sarebbe un peccato, tuttavia, affrontare la difficoltà di creare una classe e poi sforzarsi di crearne una nuova di zecca con funzionalità simili: è certamente più pratico clonare la classe esistente ed eseguire poi le aggiunte e le modifiche sulla replica. Questo è esattamente quanto si ottiene grazie all'*ereditarietà*, con l'eccezione che se la classe originale (chiamata *classe di base*, *superclasse* o *classe superiore/genitore*) subisce modifiche, la replica (chiamata *classe derivata*, *sottoclasse* o *classe figlia*) rifletterà tali cambiamenti.

5. La maggior parte dei diagrammi UML presenta dettagli sufficienti, ed è quindi spesso superfluo essere così precisi nell'indicare l'utilizzo di un'aggregazione o di una composizione.



Notate che la freccia in questo diagramma UML punta dalla classe derivata alla classe di base: come vedrete, però, comunemente esistono più classi derivate.

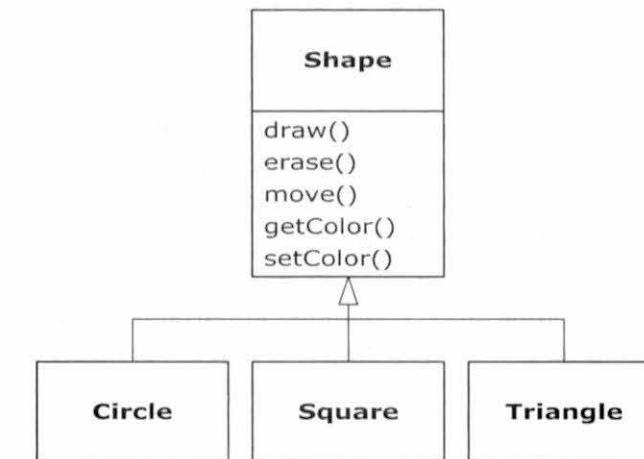
Un tipo non si limita a descrivere i vincoli esistenti su un insieme di oggetti, ma ha anche una relazione con altri tipi. Due tipi possono condividere caratteristiche e comportamenti, ma un tipo può contenere più caratteristiche di un altro, gestire più messaggi o trattarli diversamente. L'ereditarietà esprime questa somiglianza tra i tipi utilizzando i concetti di *tipi di base* e *tipi derivati*. Un tipo di base contiene tutte le caratteristiche e i comportamenti che accomunano i tipi che da esso derivano. Creerete un tipo di base per rappresentare il nocciolo delle vostre idee su alcuni oggetti nel vostro sistema; dal tipo di base deriverete poi gli altri tipi, che esprimeranno i vari modi in cui queste idee di base possono essere realizzate.

Considerate per esempio un macchinario per il riciclaggio dei rifiuti che separa i vari tipi di scarti. Il tipo di base è “rifiuti”, e ogni frammento di scorie è contraddistinto da un peso, un valore, e così via, e può essere tritato, fuso o decomposto. Da questo tipo di base derivano tipi di rifiuti più specifici, che possono avere caratteristiche o comportamenti aggiuntivi: una bottiglia ha un colore, una lattina di alluminio può essere frantumata, un contenitore di acciaio può essere magnetico. In aggiunta, alcuni comportamenti possono essere diversi: per esempio il valore della carta dipende dal tipo e dalla sua condizione. Utilizzando l'ereditarietà è possibile costruire una gerarchia di tipi che esprime il problema da risolvere in base ai tipi stessi.

Un secondo esempio è quello classico delle forme, che potrebbe essere usato nella simulazione di sistemi CAD o di giochi. Il tipo di base è costituito da oggetti **Shape** (forma), ciascuno dei quali possiede una dimensione, un colore, una posizione ecc. Ogni forma può essere tracciata, cancellata, spostata, colorata e così via. Da questo tipo di base vengono derivati (ereditati) i tipi specifici di forma, quali cerchio (**Circle**), quadrato (**Square**), triangolo



(**Triangle**), che potranno avere caratteristiche e comportamenti aggiuntivi: determinate forme possono essere capovolte e alcuni comportamenti essere diversi, per esempio quando occorra calcolare l'area di una forma. La gerarchia dei tipi incorpora quindi sia le somiglianze sia le differenze tra le forme.



Dedurre la soluzione con gli stessi termini del problema è una tecnica molto utile poiché non richiede molti modelli intermedi per passare dalla descrizione del problema a quella della soluzione. Con gli oggetti la gerarchia di tipi è il modello principale, cosicché si può passare direttamente dalla descrizione del sistema nel mondo reale alla descrizione del sistema in codice. In effetti, una difficoltà che si riscontra spesso nella progettazione OOP è rappresentata dal suo essere “troppo semplice” da portare a termine: una mente educata a ricercare soluzioni laboriose può essere inizialmente fuorviata da questa semplicità.

Ereditando da un tipo esistente si crea un nuovo tipo, che non soltanto contiene tutti i membri del tipo esistente, benché quelli **private** siano nascosti e inaccessibili, ma soprattutto duplica l'interfaccia della classe di base. In altre parole, tutti i messaggi inviabili agli oggetti della classe di base possono esserlo anche agli oggetti della classe derivata. Dal momento che il tipo di una classe può essere desunto dai messaggi che è possibile inviare, questo implica che la classe derivata è dello stesso tipo della classe di base: nell'esempio precedente, quindi, si può affermare che “un cerchio è una forma”.

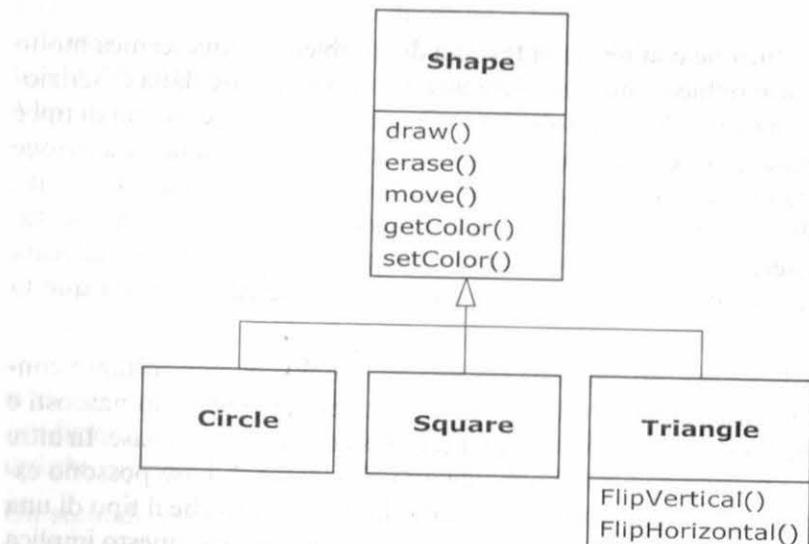
Questa equivalenza dei tipi per mezzo dell'ereditarietà è uno dei presupposti fondamentali per la comprensione della programmazione a oggetti.

Poiché sia la classe di base sia quella derivata possiedono la medesima interfaccia di base, deve esistere qualche implementazione che la accompagni: in pratica, deve esserci del codice che viene eseguito quando un oggetto riceve un particolare messaggio. Quando ci si limita a ereditare da una classe senza altri interventi, gli stessi metodi disponibili nell'interfaccia della classe di base vengono trasferiti alla classe derivata.

Questo significa che gli oggetti della classe derivata hanno non solo lo stesso tipo, ma anche lo stesso comportamento, il che non è particolarmente utile.

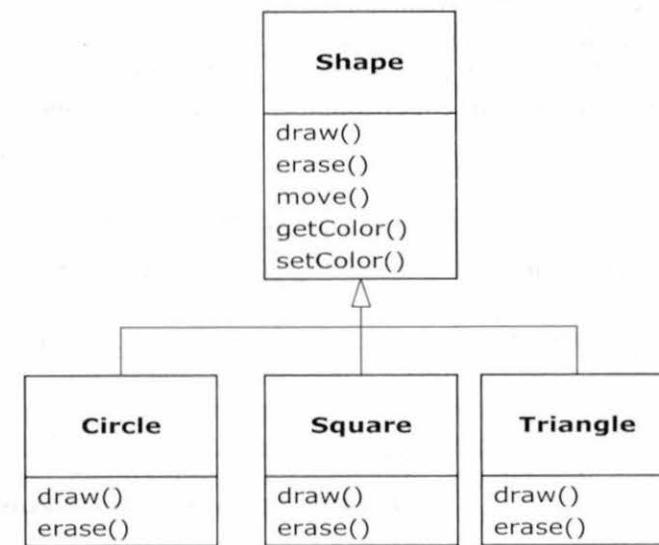
Vi sono due metodi per differenziare la nuova classe derivata dalla classe di base. Il primo è molto diretto: basta aggiungere nuovi metodi alla classe derivata, che pertanto non faranno parte dell'interfaccia della superclasse. Questo implica che la classe di base semplicemente non possedeva funzionalità sufficienti, quindi sono stati integrati altri metodi.

Questo ricorso semplice e “primitivo” all'ereditarietà è talvolta la soluzione perfetta per alcuni problemi. È comunque importante valutare con attenzione la possibilità di fornire questi metodi aggiuntivi anche alla classe di base. Questo processo di scoperta e ripetizione della progettazione si verifica regolarmente nella programmazione a oggetti.



Sebbene l'ereditarietà richieda talvolta l'aggiunta di nuovi metodi all'interfaccia (specialmente in Java, in cui la parola chiave per l'ereditarietà è **extends**) ciò non è sempre vero. Il secondo metodo, più importante, per dif-

ferenziare la nuova classe consiste nel *modificare* il comportamento di un metodo esistente nella superclasse, con una tecnica definita *sovrascrittura (overriding)* del metodo.

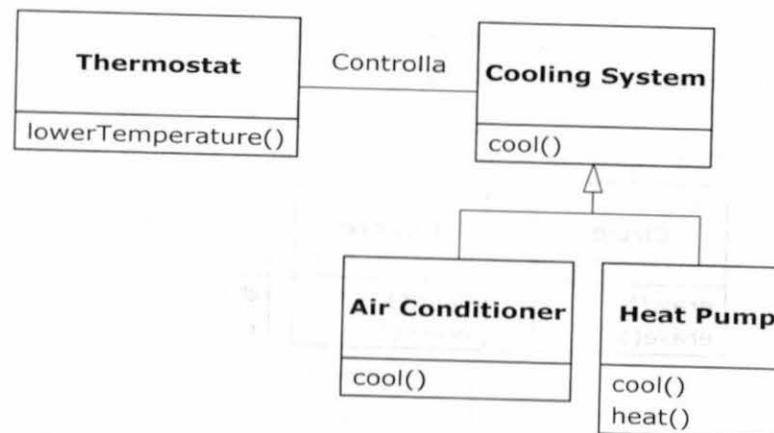


Per sovrascrivere un metodo è sufficiente creare una nuova definizione nella classe derivata; così facendo è come se diceste: “in questo punto voglio utilizzare lo stesso metodo di interfaccia, ma desidero che il nuovo tipo faccia qualcosa di diverso”.

Relazioni is-a e is-like-a

L'ereditarietà è argomento piuttosto controverso: essa non dovrebbe sovrascrivere solo i metodi della superclasse, senza aggiungerne di nuovi non presenti nella classe di base? Ciò significherebbe che la classe derivata è esattamente dello stesso tipo della superclasse, in quanto possiede la sua stessa interfaccia: sarà quindi possibile sostituire fedelmente un oggetto della sottoclasse con un oggetto della superclasse. Questo concetto può essere assimilato a una *sostituzione pura* e viene spesso definito come *principio di sostituzione*. Da un certo punto di vista questo è il modo ideale di considerare l'ereditarietà. Spesso ci si riferisce al rapporto tra la superclasse e le sottoclassi come a una relazione di tipo *is-a* (letteralmente “è-un”), poiché è possibile affermare che “un cerchio è *una* forma”. Un test per l'ereditarietà consiste nel determinare se una relazione di questo tipo tra classi è significativa.

Vi sono situazioni in cui è indispensabile aggiungere nuovi elementi di interfaccia a un tipo derivato, estendendo la sua interfaccia: il nuovo tipo può essere ancora sostituito con il tipo di base, tuttavia tale sostituzione non è perfetta perché i nuovi metodi non sono accessibili dal tipo di base. Questa può essere descritta come una relazione di tipo *is-like-a* (termine dell'autore), vale a dire “è-simile-a”. Il nuovo tipo ha l'interfaccia del tipo di base, tuttavia contiene anche altri metodi, cosicché non è perfettamente corrispondente. Considerate l'esempio di un condizionatore d'aria (**Air Conditioner**), cui si riferisce lo schema seguente.



Supponete che la vostra abitazione sia adeguatamente cablata e che disponiate di un'interfaccia per controllarne il raffreddamento. Immaginate che il condizionatore (**Air Conditioner**) si guasti e che lo sostituiate con un climatizzatore a pompa di calore (**Heat Pump**), in grado sia di raffreddare sia di riscaldare l'ambiente. In questo caso il climatizzatore sarà simile a un condizionatore, ma dotato di funzionalità aggiuntive.

Tenuto conto che il sistema di controllo della vostra abitazione è stato progettato per il solo raffreddamento, si limiterà alla comunicazione con la parte refrigerante del nuovo oggetto. L'interfaccia del nuovo oggetto sarà stata estesa, certo, tuttavia il sistema esistente continuerà a riconoscere soltanto l'interfaccia originale.

Una rapida occhiata a questo schema evidenzia che la classe di base **Cooling System** (sistema di raffreddamento) non è abbastanza generica, e dovrebbe essere rinominata in qualcosa di simile a “sistema di controllo della temperatura”, così da poter includere anche il riscaldamento. È soltanto a questo punto che il principio di sostituzione potrà funzionare. Comunque sia, que-

sto diagramma è un esempio tipico di quanto può avvenire realmente nella progettazione.

Quando si può ricorrere al principio di sostituzione è facile ritenere che questo approccio di sostituzione pura sia l'unico modo di procedere; in effetti risulta pratico quando la progettazione lo consente. Tuttavia, vedrete che vi sono situazioni in cui è ugualmente chiara la necessità di aggiungere nuovi metodi all'interfaccia di una classe derivata. Con un'analisi accurata entrambi i casi dovrebbero essere ragionevolmente evidenti.

Intercambiabilità degli oggetti mediante polimorfismo

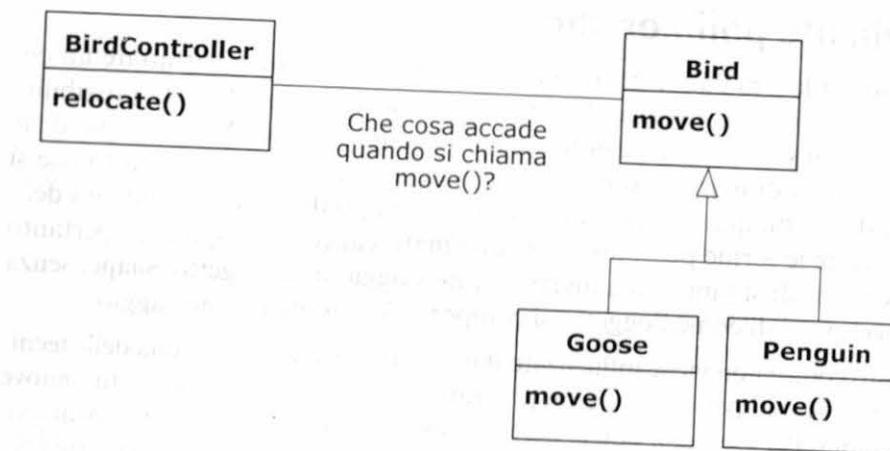
Quando si lavora con le gerarchie di tipi spesso non si vuole trattare un oggetto come il tipo specifico, bensì come il suo tipo base. Questa possibilità permette di scrivere codice indipendente dai tipi specifici. Nell'esempio delle forme i metodi manipolano quelle generiche, a prescindere dal fatto che si tratti di cerchi, quadrati, triangoli o un altro tipo di forma non ancora definita. Tutte le forme possono essere disegnate, cancellate e spostate, pertanto questi metodi si limitano a inviare un messaggio a un oggetto **Shape**, senza preoccuparsi di come l'oggetto si comporterà a fronte del messaggio.

Questo codice non viene influenzato dall'aggiunta di nuovi tipi, una delle tecniche più comuni per estendere un programma OOP allo scopo di gestire nuove situazioni. Per esempio, potete derivare un nuovo sottotipo di forma pentagonale (**Pentagon**) senza modificare i metodi che si occupano solo delle forme generiche. Tale capacità di estendere facilmente un progetto derivando nuovi sottotipi è uno dei metodi essenziali per encapsulare il cambiamento, che migliora sensibilmente la progettazione e riduce il costo di manutenzione del software.

Esiste tuttavia un problema sostanziale quando si cerca di trattare gli oggetti derivati come i loro rispettivi tipi di base generici (cerchi come forme, biciclette come veicoli, cormorani come uccelli ecc.). Se un metodo ordina a una forma generica di disegnarsi, a un veicolo generico di sterzare, o a un uccello generico di spostarsi, il compilatore non può sapere in fase di compilazione quale parte di codice sarà esattamente in esecuzione. Questo è il punto: quando il messaggio viene inviato il programmatore non vuole sapere quale porzione di codice sarà eseguita; il metodo di disegno **draw()** può essere applicato indifferentemente a un cerchio, un quadrato o un triangolo, e l'oggetto eseguirà il codice corretto in base al suo tipo specifico.

Se non è indispensabile conoscere la porzione di codice che sarà eseguita, allora quando si aggiunge un nuovo sottotipo il codice eseguito potrà essere di-

verso, senza richiedere cambiamenti al metodo che lo richiama. Il compilatore non può sapere con precisione quale parte di codice viene eseguita, quindi a questo punto è necessario chiedersi "che cosa fa" il compilatore. Per esempio, nel diagramma seguente l'oggetto **BirdController** opera solo con oggetti **Bird** generici, senza sapere esattamente di quale uccello si tratti. Questo è pratico dal punto di vista di **BirdController**, perché non è necessario scrivere codice particolare per determinare il tipo esatto di uccello con cui si sta lavorando, né il suo comportamento specifico. Occorre ora chiedersi come è possibile che, chiamando il metodo **move()** per lo spostamento, pur ignorando il tipo specifico di **Bird** si verificherà comunque il comportamento corretto: un'oca (**Goose**) cammina, vola o nuota e un pinguino (**Penguin**) cammina o nuota.



La risposta rappresenta la svolta principale introdotta dalla programmazione a oggetti: il compilatore non può eseguire una chiamata a una funzione nel senso tradizionale del termine. La chiamata di funzione generata da un compilatore non-OOP genera il cosiddetto *early binding* o *binding anticipato* (noto anche come *binding statico* o *a tempo di compilazione*), termini che potrebbero essere nuovi al lettore che non ha mai pensato esista un diverso tipo di compilazione. Queste espressioni indicano che il compilatore genera una chiamata a un nome di funzione specifico; il sistema, in fase di esecuzione, risolve questa chiamata nell'indirizzo assoluto del codice da eseguire. Nella programmazione OOP il programma non può determinare l'indirizzo del codice fino al momento dell'esecuzione, quindi, se un messaggio è inviato a un oggetto generico, è necessario adottare uno schema diverso.

Per risolvere il problema, i linguaggi orientati agli oggetti utilizzano il concetto di *late binding* o *binding ritardato* (noto anche come *binding dinamico* o *a tempo*

di esecuzione). Quando si invia un messaggio a un oggetto, il codice chiamato non viene determinato fino al momento dell'esecuzione. Il compilatore si assicura che il metodo esista, esegue una verifica del tipo controllando gli argomenti e il valore di ritorno, tuttavia non sa qual è il codice esatto da eseguire.

Per eseguire il binding ritardato Java utilizza una porzione di codice particolare, che sostituisce la chiamata assoluta. Questo codice calcola l'indirizzo del corpo del metodo, utilizzando le informazioni registrate nell'oggetto stesso, secondo un meccanismo che sarà spiegato in dettaglio nel Capitolo 8 dedicato al polimorfismo. In questo modo ogni oggetto può comportarsi diversamente in base al contenuto di questo particolare frammento di codice. Quando inviate un messaggio a un oggetto, esso in effetti riesce a capire che cosa deve fare con quel messaggio.

In alcuni linguaggi è necessario dichiarare esplicitamente di voler utilizzare un metodo dotato della flessibilità offerta dal binding ritardato: il C++, per esempio, richiede l'uso della parola chiave **virtual**. In modalità predefinita, in questi linguaggi i metodi non sono legati dinamicamente; in Java, invece, il binding ritardato costituisce il comportamento predefinito, di conseguenza non è necessario ricordare parole chiave supplementari per usufruire del polimorfismo.

Considerate di nuovo l'esempio delle forme: la famiglia di classi, interamente basata sulla stessa interfaccia uniforme, è già stata schematizzata nei precedenti diagrammi. Per dimostrare il polimorfismo, supponete di voler scrivere una porzione di codice che ignori i dettagli specifici del tipo e comunichi unicamente con il codice della classe di base. Questo codice è *svincolato* dalle informazioni specifiche del tipo, pertanto risulterà più semplice da scrivere e più facile da comprendere. Inoltre, qualora aggiungiate un nuovo tipo, per esempio **Hexagon**, per mezzo dell'ereditarietà, il codice funzionerà ugualmente con il nuovo tipo di forma esattamente come avviene per i tipi esistenti: in questo caso, si dice che il programma è *estendibile*.

Scrivendo un metodo Java, come

```

void doSomething(Shape shape) {
    shape.erase();
    // ...
    shape.draw();
}
  
```

otterrete che questo metodo comunica con qualsiasi oggetto **Shape**, ed è quindi indipendente dal tipo specifico dell'oggetto che si sta tracciando con



draw() o cancellando con **erase()**. Se un'altra parte del programma usasse il metodo **doSomething()**:

```
Circle circle = new Circle();
Triangle triangle = new Triangle();
Line line = new Line();
doSomething(circle);
doSomething(triangle);
doSomething(line);
```

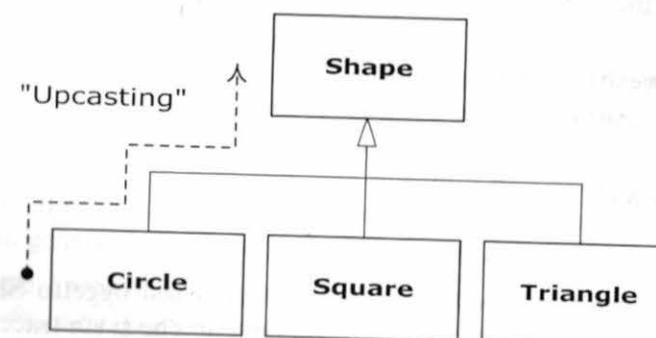
la chiamata a **doSomething()** opererà automaticamente nel modo corretto, a prescindere dal tipo esatto di oggetto.

Questo è effettivamente un trucco sorprendente. Considerate la riga:

```
doSomething(circle);
```

Ciò che accade in questo caso è che un oggetto **Circle** viene passato a un metodo che si attende uno **Shape**. Poiché **Circle** è uno **Shape**, può essere trattato come tale da **doSomething()**. In pratica, qualsiasi messaggio che **doSomething()** possa inviare a uno **Shape** potrà essere accettato da un **Circle**, il che rappresenta quanto di più logico ci si possa aspettare.

Questa tecnica che permette di trattare un tipo derivato come se fosse il suo tipo di base è detta *upcasting*. Il termine *casting* è utilizzato nel senso di "colare, replicare in uno stampo", mentre il prefisso *up* deriva dalla tipica organizzazione dei diagrammi UML di ereditarietà, nei quali il tipo di base è in alto e le classi derivate si diramano verso il basso. Quindi, il casting verso un tipo di base comporta uno spostamento verso l'alto nel diagramma: un *upcasting*, appunto.



Un programma orientato agli oggetti contiene, in qualche punto, operazioni di upcasting, poiché questo è il modo per "svincolarsi" dalla necessità di conoscere il tipo esatto con cui si sta operando. Osservate il codice del metodo **doSomething()**:

```
shape.erase();
// ...
shape.draw();
```

Noteate che questo codice non esprime il concetto: "se sei un **Circle** esegui questa operazione, se sei uno **Square** fai quest'altra e così via". Questo tipo di codice, che controllerebbe tutti i tipi che uno **Shape** può assumere, sarebbe confuso e richiederebbe di essere modificato ogni volta che aggiungete una nuova forma. Invece vi limitate a dire: "sei uno **Shape**, so che puoi eseguire **erase()** e **draw()**, fallo e occupati dei dettagli corretti".

Nel metodo **doSomething()** è interessante notare che, in qualche modo, viene sempre eseguita l'operazione corretta. La chiamata a **draw()** per **Circle** fa in modo che sia eseguito codice diverso rispetto alla chiamata di **draw()** per **Square** o **Line**; tuttavia, quando il messaggio **draw()** viene inviato a una **Shape** anonima, il comportamento corretto si basa sul tipo effettivo dello **Shape**. Questo meccanismo è sorprendente perché, come si è detto, quando il compilatore Java compila il codice di **doSomething()** non può sapere esattamente con quali tipi dovrà operare. Di norma, quindi, si potrebbe prevedere la chiamata delle versioni di **erase()** e **draw()** relative alla classe di base **Shape**, non quella per i tipi specifici **Circle**, **Square** o **Line**.

È grazie al polimorfismo che viene eseguita l'operazione giusta: il compilatore e il sistema di runtime gestiscono automaticamente tutti i dettagli; per il momento, vi basti sapere che questo accade e come servirvi di questi meccanismi nella progettazione. Quando inviate un messaggio a un oggetto, questo eseguirà l'operazione corretta, anche quando interviene l'upcasting.

Gerarchia a radice comune

Una delle considerazioni della programmazione OOP che è emersa soprattutto dall'introduzione del linguaggio C++, è che tutte le classi dovrebbero, in ultima analisi, derivare da una sola classe di base. Questo è vero per quasi tutti i linguaggi OOP, tranne C++: in Java questa classe di base è chiamata semplicemente **Object**. I vantaggi offerti da questa gerarchia a radice comune sono numerosi.



Tutti gli oggetti presenti in una gerarchia a radice comune condividono un'interfaccia, cosicché, in sostanza, riproducono lo stesso tipo fondamentale. L'alternativa fornita da C++ è "non sapere che tutto rimonta allo stesso tipo di base". Dal punto di vista della retrocompatibilità questo approccio si integra meglio con il modello C, e può essere considerato meno restrittivo; tuttavia, quando occorre utilizzare una programmazione completamente orientata agli oggetti, è necessario sviluppare una propria gerarchia per sfruttare le stesse caratteristiche tipiche degli altri linguaggi a oggetti.

Come se non bastasse, in ogni nuova libreria di classi che venga acquisita possono essere presenti altre interfacce incompatibili: questo impone uno sforzo maggiore e può richiedere anche l'ereditarietà multipla, al fine di adattare la nuova interfaccia nel progetto. La "flessibilità" aggiuntiva di C++ è davvero meritevole di una simile attività supplementare? Se siete vincolati a questo linguaggio, magari perché avete una grande quantità di codice C esistente, può certamente avere un grande valore; se invece iniziate *ex novo*, alternative come Java risultano spesso più produttive.

Tutti gli oggetti in una gerarchia a radice unica possiedono sicuramente determinate funzionalità. Inoltre, si sa che è possibile eseguire una certa serie di operazioni di base su tutti gli oggetti del sistema. Tutti gli oggetti possono essere generati facilmente nella memoria *heap* (come si vedrà più avanti) e il passaggio degli argomenti è notevolmente semplificato.

Una gerarchia a radice comune rende molto più semplice la creazione di un *garbage collector*, che costituisce uno dei miglioramenti fondamentali di Java rispetto a C++. Inoltre, poiché in tutti gli oggetti è garantita la presenza di informazioni sul tipo di oggetto, non è possibile ritrovarsi con un oggetto di cui non sia determinabile il tipo. Questa caratteristica è importante soprattutto nelle operazioni a livello di sistema, quali la gestione delle eccezioni, e per garantire una maggiore flessibilità in fase di programmazione.

Contenitori

In generale è impossibile sapere quanti oggetti sono necessari per risolvere un problema specifico, né quanto tempo dureranno. Per di più, non è possibile sapere neppure come memorizzare questi oggetti. Del resto, come fare per conoscere quanto spazio creare se queste informazioni sono note soltanto in fase di esecuzione?

La soluzione alla maggior parte dei problemi di progettazione OOP potrà apparire addirittura poco seria: creare un altro tipo di oggetto. Il nuovo tipo



di oggetto che risolve questo specifico problema mantiene i riferimenti ad altri oggetti.

Naturalmente potete ottenere lo stesso risultato con un *array*, disponibile nella maggior parte dei linguaggi. Questo nuovo oggetto è solitamente chiamato contenitore (*container*) o collezione (*collection*): tenete presente, tuttavia, che nella libreria Java quest'ultimo termine è utilizzato con un'accezione diversa, pertanto in questo volume si è scelto di usare il primo. Tale oggetto si espanderà ogni volta che ciò sia necessario, per accogliere quanto vi sarà inserito; così facendo non sarà necessario conoscere a priori quanti oggetti verranno inseriti in un contenitore, ma basterà creare un oggetto contenitore che si occuperà dei dettagli autonomamente.

Per fortuna un buon linguaggio OOP è fornito con un insieme di contenitori come parte del pacchetto. In C++ questa funzionalità fa parte della *Standard C++ Library* e viene spesso chiamata *Standard Template Library (STL)*. Smalltalk possiede un insieme molto completo di container, così come la libreria standard di Java. In alcune librerie, uno o due contenitori generici sono considerati sufficienti per ogni esigenza; in altri linguaggi, per esempio Java, la libreria fornisce vari tipi di contenitori per le diverse necessità: alcuni generi di liste (classi **List**, pensate per contenere sequenze), mappe (**Map**, conosciute anche come *array associativi*, con cui è possibile associare oggetti ad altri oggetti), insiemi (**Set**, per contenere un esemplare di ciascun tipo di oggetto) e altri componenti quali code, alberature, stack ecc.

Dal punto di vista della progettazione, tutto ciò che occorre realmente è un contenitore che possa essere manipolato per risolvere il problema. Se un solo tipo di contenitore bastasse a soddisfare tutte le necessità, non sarebbe necessario averne diversi. Due sono le ragioni per cui è opportuno disporre di una scelta di contenitori così vasta. Innanzitutto essi mettono a disposizione vari tipi di interfacce e comportamenti esterni: uno stack ha comportamento e interfaccia diversi da una coda, che a sua volta differisce da un insieme o da una lista. Uno di questi elementi potrebbe rivelarsi più adatto di un altro per risolvere un problema.

In secondo luogo, contenitori differenti hanno efficienza diversa per alcune operazioni. Per esempio vi sono due tipi fondamentali di liste, **ArrayList** e **LinkedList**, in entrambi i casi semplici sequenze che possono avere interfacce e comportamenti esterni identici. Determinate operazioni, tuttavia, possono avere costi, in termini di CPU e risorse, notevolmente diversi: in particolare l'accesso casuale agli elementi contenuti in un **ArrayList** richiede un tempo costante a prescindere dall'elemento selezionato; in una **LinkedList**, invece, è oneroso spostarsi nella lista per selezionare un elemento a caso, e occorre più tempo per trovare un elemento in posizione più distante. D'altro canto,

se si vuole inserire un elemento nella parte centrale di una sequenza è meno oneroso farlo in una **LinkedList** che in un **ArrayList**.

Queste e altre operazioni sono caratterizzate da un diverso grado di efficienza, in funzione della struttura su cui è basata la sequenza utilizzata. Nulla vieta, in ogni caso, di iniziare lo sviluppo di un programma con una **LinkedList** e, dopo aver valutato le prestazioni, passare a un **ArrayList**. Grazie all'astrazione offerta dall'interfaccia **List** è possibile passare da una tecnica all'altra con un impatto minimo sul codice.

Tipi parametrizzati o generici

Prima di Java SE5 i contenitori contenevano il solo tipo universale di Java: **Object**. La gerarchia a radice comune indica che tutto è un **Object**, pertanto un contenitore che ingloba oggetti **Object** può contenere qualsiasi elemento.⁶

Questo ha semplificato il riutilizzo dei contenitori. Per usare un contenitore era sufficiente aggiungervi i riferimenti degli oggetti, che potevano poi essere estratti a richiesta. Purtroppo, però, dal momento che il contenitore conteneva soltanto oggetti di tipo **Object**, aggiungendo il riferimento dell'oggetto questo subiva un upcasting a **Object**, con conseguente perdita delle proprie caratteristiche. Al momento dell'estrazione dei dati si otteneva un riferimento a un oggetto **Object**, non quello relativo al tipo di oggetto inserito. Il problema, pertanto, consisteva nel trasformare di nuovo questo riferimento nel tipo specifico di oggetto che era stato inserito in origine.

Anche in questo caso si ricorre a una tecnica di casting, questa volta non per cercare di risalire la gerarchia ereditaria allo scopo di ottenere un tipo più generico, bensì per discendere la gerarchia verso un tipo più specifico: tale meccanismo di casting è chiamato *downcasting*. Con l'*upcasting* si sa con certezza, per esempio, che un **Circle** è di tipo **Shape**, pertanto l'esecuzione dell'*upcasting* non dà luogo a problemi; non è invece possibile determinare se un **Object** sia un **Circle** oppure un oggetto **Shape**: per questa ragione può essere rischioso eseguire il *downcasting*, a meno di non sapere esattamente ciò che si sta facendo.

Tenete presente, in ogni caso, che tale rischio è in qualche modo attenuato, perché eseguendo il *downcasting* al tipo errato si ottiene un errore di runtime, denominato *eccezione*, che sarà descritto tra breve. Quando si estraggono i riferimenti degli oggetti da un contenitore, tuttavia, è necessario disporre

6. Non può contenere i cosiddetti tipi primitivi; tuttavia, come vedrete in dettaglio più avanti, la funzionalità autoboxing di Java SE5 rende pressoché nulla questa limitazione.

di un metodo per ricordare esattamente il loro tipo, al fine di eseguire poi il *downcasting* appropriato.

Il *downcasting* e i controlli a runtime incrementano i tempi di esecuzione del programma e richiedono uno sforzo supplementare da parte del programmatore. Non sarebbe quindi meglio generare il contenitore in modo che riconosca i tipi da incorporare, eliminando la necessità del *downcasting* e la possibilità che si verifichino errori? La risposta a questa domanda è nel meccanismo denominato *tipo parametrizzato*, una classe che il compilatore può adattare automaticamente per operare con tipi particolari. Per esempio, il compilatore potrebbe adattare un contenitore parametrizzato in modo che accetti e restituisca soltanto oggetti di tipo **Shape**.

Una delle innovazioni principali di Java SE5 è costituita dall'aggiunta dei tipi parametrizzati, che Java definisce con il termine di generici (*generics*). L'utilizzo dei generici è riconoscibile dall'indicazione dei tipi inclusa nelle cosiddette "parentesi angolate", vale a dire la coppia di simboli minore (<) e maggiore (>). Per esempio, un **ArrayList** contenente oggetti **Shape** viene generato come segue:

```
ArrayList<Shape> shapes = new ArrayList<Shape>();
```

Per sfruttare i generici, in Java SE5 sono state apportate modifiche in molti componenti della libreria standard. Come vedrete, i generici hanno impatto su buona parte del codice contenuto in questo manuale.

Creazione e durata degli oggetti

Un argomento critico della programmazione OOP è il modo in cui gli oggetti vengono creati e distrutti. L'esistenza di ogni oggetto richiede risorse, in particolare se collocato in memoria: di conseguenza, quando un oggetto non è più necessario deve essere eliminato, in modo da rilasciare le risorse e consentire il loro riutilizzo. Nei progetti più semplici, il modo in cui un oggetto viene eliminato non è particolarmente stimolante: l'oggetto viene generato, usato finché è necessario, poi dovrebbe essere distrutto. Tuttavia, non è difficile incontrare situazioni ben più complesse.

Supponete, per esempio, di progettare un sistema per la gestione del traffico aeroportuale: lo stesso modello sarebbe valido anche per la movimentazione automatica di un magazzino, per un sistema di videonoleggio o per la gestione di una pensione per animali. A prima vista sembra semplice: si genera un contenitore per contenere i velivoli, poi si crea un nuovo velivolo, inserendo



nel contenitore ogni velivolo che si immetta nella zona di controllo del traffico aereo. Per "ripulire" la memoria è sufficiente eliminare l'oggetto velivolo appropriato, quando l'aeroplano cui si riferisce lascia la zona di controllo.

È possibile, tuttavia, che esista un altro sistema per la registrazione dei dati sugli aeroplani; potrebbe trattarsi di dati che non richiedono un'attenzione così immediata come la funzione di controllo principale, per esempio i piani di volo dei piccoli velivoli che lasciano l'aeroporto. Quindi, si creerà un secondo contenitore destinato ai velivoli di piccole dimensioni: ogniqualvolta si crea un oggetto velivolo, se relativo a un aeroplano di piccole dimensioni esso verrà inserito anche in questo secondo contenitore. In seguito, durante i periodi di inattività del computer, un processo in background si incaricherà di eseguire le operazioni opportune sugli oggetti presenti in questo contenitore.

Ora il problema diventa più difficile: come fare per sapere quando distruggere gli oggetti? Quando l'utente ha terminato di lavorare con l'oggetto, un'altra parte del sistema potrebbe non aver completato l'elaborazione. Questo stesso problema può sorgere in molte altre situazioni e negli ambienti di sviluppo, come il C++, in cui occorre cancellare esplicitamente un oggetto quando non più necessario, può anche diventare piuttosto complesso.

Dove sono conservati i dati di un oggetto, e come viene controllato il suo ciclo di vita? Il linguaggio C++ parte dal presupposto che il controllo dell'efficienza sia la caratteristica più importante, per cui offre al programmatore una scelta. Per ottenere la massima velocità di esecuzione, la registrazione e il ciclo di vita possono essere determinati durante la stesura del programma, collocando gli oggetti nello stack (in quelle che vengono talvolta indicate come *variabili automatiche* o *scoped*) oppure nell'area di memorizzazione statica. Questa tecnica impone una priorità sulla velocità di allocazione e rilascio della memoria, un tipo di controllo che può essere utile in molte situazioni. Tuttavia, questo approccio a favore della velocità va a detimento della flessibilità, perché richiede che durante la scrittura del programma siano note la quantità, il ciclo di vita e l'esatto tipo degli oggetti: diventa perciò troppo restrittivo per risolvere problemi più generici, per esempio nelle applicazioni CAD, di gestione magazzino o di controllo del traffico aereo.

Il secondo approccio consiste nella creazione dinamica degli oggetti in un'area di memoria detta *heap*. Mediante questo meccanismo non occorre che siano noti il numero, il ciclo di vita e il tipo di oggetti necessari fino al momento dell'esecuzione. Tali informazioni vengono determinate dalle necessità contingenti quando il programma è in esecuzione. Nel momento stesso in cui si renda necessario un nuovo oggetto, esso potrà essere creato nell'heap: tuttavia, trattandosi di memorizzazione dinamica, ossia gestita



durante l'esecuzione, il tempo necessario per allocare spazio nell'heap può essere notevolmente superiore a quello richiesto dallo stack. La creazione di spazio di archiviazione nello stack consiste di solito in un'istruzione in Assembler, per spostare il puntatore dello stack verso il basso, e di una seconda istruzione per spostarlo verso l'alto. Invece, il tempo necessario per creare spazio nell'heap dipende dalla struttura del meccanismo di archiviazione.

L'approccio dinamico si basa sull'ipotesi, generalmente logica, che gli oggetti hanno tendenza a essere complessi, cosicché il costo gestionale supplementare per la memorizzazione e il rilascio degli oggetti non ha un impatto determinante sulla creazione di un oggetto. Inoltre, la maggiore flessibilità è essenziale per risolvere i problemi di programmazione generici.

Java utilizza esclusivamente l'allocazione della memoria dinamica: ogni volta che si desidera creare un oggetto, si usa l'operatore **new** per costruire un'istanza dinamica dell'oggetto.⁷

Un altro argomento merita attenzione: la durata di un oggetto. Con i linguaggi che permettono di creare gli oggetti nello stack, il compilatore determina la durata dell'oggetto e può distruggerlo automaticamente. Tuttavia, quando si crea l'oggetto nell'heap, il compilatore non dispone di informazioni sul suo ciclo di vita. In un linguaggio come il C++ bisogna determinare in modo programmatico quando distruggere l'oggetto, e questo può dare luogo alle cosiddette "perdite di memoria" (*memory leaks*) se l'operazione non viene eseguita correttamente: un problema comune a molti programmi scritti in C++. Java, di contro, fornisce una funzionalità detta *garbage collector* che rileva automaticamente un oggetto non più in uso e lo distrugge. Si tratta di un meccanismo ben più pratico, in quanto riduce il numero di argomenti da seguire e il codice da scrivere: ma soprattutto il garbage collector garantisce un elevato livello di sicurezza a fronte dell'insidioso problema dei memory leak che, da solo, ha messo in ginocchio numerosi progetti C++.

In Java il garbage collector è concepito per occuparsi del rilascio della memoria, benché questo non includa altri aspetti riferiti al cleanup di un oggetto. Il garbage collector "sa" quando un oggetto non è più in uso e rilascia automaticamente la memoria da esso impegnata. Questa caratteristica, combinata con il fatto che tutti gli oggetti vengono ereditati dall'unica singola classe radice **Object**, e che esiste un solo modo di creare oggetti (nell'heap), rende la programmazione Java assai più semplice di quella C++: meno decisioni da prendere e meno ostacoli da superare.

7. I tipi primitivi, come vedrete in seguito, rappresentano un'eccezione.



Gestione delle eccezioni e trattamento degli errori

Già nei primi linguaggi di programmazione la gestione degli errori è stata un'attività particolarmente complessa da gestire. Poiché è così difficile progettare un valido sistema di gestione degli errori, molti linguaggi non fanno altro che ignorare l'argomento, trasferendo il problema ai progettisti delle librerie; questi si limitano a implementare misure intermedie che funzionano bene in molte situazioni, ma che possono anche essere raggirate, semplicemente ignorandole. Un problema comune alla maggior parte degli schemi di gestione errori risiede nel fatto che essi si basano sull'attenzione del programmatore e sul suo scrupolo nell'attenersi a una convenzione concordata, che non è imposta dal linguaggio. Se lo sviluppatore non è attento, o come spesso accade, se si programma di fretta, questo schema viene facilmente dimenticato.

La gestione delle eccezioni ingloba il trattamento degli errori direttamente nel linguaggio di programmazione, talvolta persino nel sistema operativo. Un'eccezione è un oggetto che "viene sollevato" nel punto in cui avviene l'errore, e che può essere "intercettato" da un apposito gestore di eccezioni destinato a trattare quel particolare tipo di errore. Il trattamento delle eccezioni si comporta come un percorso differente e parallelo dell'esecuzione, che può essere preso quando si verifica una situazione anomala. Poiché è un percorso di esecuzione separato, non interferisce con la normale esecuzione del codice: questo rende il codice più semplice da scrivere, perché non si è costretti costantemente a verificare la presenza di errori. Inoltre, il sollevamento di un'eccezione è un meccanismo diverso dalla restituzione di un valore di errore da parte di un metodo, e dall'impostazione di un flag di controllo da parte di un metodo per indicare una condizione di errore: questi sistemi possono essere banalmente ignorati.

Un'eccezione non può essere ignorata, ma richiede di essere presa in considerazione. Le eccezioni forniscono uno strumento per ovviare in modo affidabile a una situazione di errore: anziché uscire dal programma, spesso si ha la possibilità di impostare i parametri corretti e ripristinare l'esecuzione, e questo offre l'opportunità di creare programmi molto più stabili.

Java spicca tra i vari linguaggi di programmazione per la sua gestione delle eccezioni, che è implicita nel linguaggio stesso e costringe il programmatore a utilizzarla: rappresenta l'unico modo accettabile per segnalare gli errori. Se non scrivete il codice per gestire correttamente le eccezioni otterrete un messaggio di errore durante la compilazione. Questa garanzia di coerenza spesso rende la gestione degli errori notevolmente più semplice.



È opportuno osservare che la gestione delle eccezioni non è una caratteristica orientata agli oggetti, per quanto nei linguaggi orientati di tipo OOP l'eccezione sia normalmente rappresentata da un oggetto: la gestione delle eccezioni è esistita ben prima dei linguaggi a oggetti.

Programmazione concorrente

Un concetto fondamentale della programmazione è la gestione simultanea di varie operazioni. Molti problemi di programmazione richiedono che il programma interrompa l'operazione corrente, si occupi di un altro problema, per ritornare infine al processo principale. La risposta a questa esigenza è stata fornita in molti modi. Agli inizi, gli sviluppatori con conoscenza di programmazione a basso livello scrivevano routine per la gestione degli interrupt, e la sospensione del flusso principale veniva attivata da un interrupt di tipo hardware. Sebbene questo sistema funzionasse bene, era complesso e non portatile, tale da rendere la migrazione di un programma su un nuovo sistema un'operazione lenta e costosa.

A volte gli interrupt sono necessari per gestire le operazioni cosiddette *time-critical*, per le quali i tempi di risposta del sistema sono determinanti: esiste tuttavia un gran numero di situazioni in cui si vuole soltanto suddividere il problema in parti eseguite separatamente (*task*) in modo che l'intero programma offra prestazioni migliori. Nell'ambito di un programma, le singole parti eseguite in modo autonomo sono denominate *thread* e il concetto generale è chiamato *concorrenza (concurrency)*. Un tipico esempio di concorrenza è comune nelle interfacce utente: mediante i task, l'utente può premere un pulsante e ottenere una risposta senza dover attendere che il programma completi l'operazione corrente.

In genere i task non sono altro che un modo per allocare il tempo di un singolo processore: se il sistema operativo supporta processori multipli, tuttavia, ogni attività può essere assegnata a un processore diverso e tutte possono essere eseguite in modo realmente parallelo. Una delle funzionalità offerte dalla concorrenza a livello di linguaggio è che il programmatore non deve preoccuparsi del numero di processori effettivo: il programma è suddiviso logicamente in attività, e se il computer dispone di più processori il programma viene eseguito in modo più rapido, senza alcun adattamento.

Tutto questo fa sì che la concorrenza sembri un concetto piuttosto semplice da gestire. Ma, come in ogni cosa, esiste un punto debole: le risorse condivise. Se sul computer sono in esecuzione diversi task che si aspettano di accedere alla stessa risorsa, si verificherà un problema. Per esempio, due processi non possono trasmettere simultaneamente le informazioni a una stessa



stampante. Per risolvere il problema le risorse che possono essere condivise, quali le stampanti, devono essere bloccate durante il loro impiego: così facendo un task bloccherà la risorsa, completerà la sua operazione e rilascerà il blocco in modo che altri task possano accedere alla stessa risorsa. In Java la concorrenza è integrata nel linguaggio; Java SE5 ha aggiunto un significativo supporto di libreria.

Java e Internet

Se Java, in effetti, è soltanto un altro linguaggio, potreste mettere in discussione la sua importanza e il fatto che rappresenti un passo rivoluzionario nella programmazione. Questo, infatti, non è immediatamente evidente per chi proviene da un'esperienza di sviluppo tradizionale. Benché Java sia molto utile per risolvere i classici problemi della programmazione *standalone*, è importante anche perché offre la soluzione a problemi di programmazione connessi al World Wide Web.

Che cos'è il Web?

Il Web può sembrare dapprima un'entità misteriosa, per tutto questo parlare di "navigare", "presenza" e "home page". È utile fare un passo indietro e vedere di che cosa si tratta effettivamente; per fare questo, è importante comprendere i sistemi client/server, un altro aspetto dell'informatica che abbondano di argomenti fuorvianti.

Programmazione client/server

L'idea originale di un sistema client/server è quello di avere un punto di raccolta centralizzato delle informazioni, rappresentate da tipi di dato qualsiasi, generalmente ospitate in un database, che si vuole distribuire su richiesta a un insieme di persone o di sistemi. Il concetto chiave di un sistema client/server è che il deposito delle informazioni sia centralizzato, in modo che le modifiche si propaghino agli utenti. Globalmente, il deposito dei dati, il software incaricato di distribuire le informazioni e i sistemi in cui risiedono le informazioni e il software sono denominati *server*. Il software residente sul computer dell'utente comunica con il server, ottiene le informazioni, le elabora e le visualizza sul sistema utente, chiamato *client*.

Il concetto di base dell'elaborazione client/server, quindi, non è molto complesso. I problemi sorgono perché si dispone di un solo server che cerca di "servire" molti client nello stesso momento. In genere è previsto un sistema di gestione di database, affinché il progettista possa "bilanciare" la distribu-



zione dei dati nelle tabelle per ottimizzarne l'utilizzo. Inoltre, spesso i sistemi consentono a un client di inserire dati nel server: questa condizione implica l'esigenza di assicurarsi che i nuovi dati forniti da un client non sovrascrivano quelli inviati da altri client, e che i dati non vadano persi nel processo di inserimento nel database: a questo scopo si ricorre alla cosiddetta *elaborazione transazionale*. Quando il software client viene modificato deve essere compilato, sottoposto a correzioni e verifiche, e infine installato sui sistemi client: questo processo risulta più complesso e costoso di quanto possiate pensare. In particolare, uno dei problemi principali riguarda il supporto a diversi tipi di computer e sistemi operativi.

Non si devono poi dimenticare le prestazioni, un argomento di primaria importanza: potreste avere centinaia di client che eseguono richieste al server in qualsiasi momento, di conseguenza un piccolo ritardo nella risposta potrebbe costituire un fattore critico. Per ridurre al minimo i tempi di latenza i programmati si sforzano di "delegare" le operazioni di elaborazione, spesso al computer client, ma talvolta anche ad altri sistemi lato server, utilizzando il cosiddetto *middleware*, un insieme di software usato per migliorare la gestibilità nel suo complesso.

La semplice idea della distribuzione di informazioni ha così tanti livelli di complessità che l'intero problema può apparire come un enigma senza soluzione. In ogni caso si tratta di un problema imponente, tenuto conto che l'elaborazione client/server rappresenta circa il 50% di tutte le attività di programmazione: è responsabile di ogni cosa, dalla gestione degli ordini e delle transazioni con carta di credito, alla distribuzione di dati finanziari, azionari, scientifici, governativi e via dicendo. Ciò che ha contraddistinto la storia della programmazione client/server sono singole soluzioni a singoli problemi, inventando di continuo nuove soluzioni difficili da creare e complesse da usare, che richiedevano che l'utente familiarizzasse ogni volta con nuove interfacce. L'intero problema client/server doveva essere risolto in modo più radicale.

Il Web come gigantesco server

Il Web è un colossale sistema client/server, caratterizzato però da un livello di complessità aggiuntivo, poiché prevede che tutti i server e i client coesistano simultaneamente su una sola rete. Di norma, questa ulteriore caratteristica è del tutto trasparente, dal momento che basta occuparsi del collegamento e dell'interazione con un server per volta, anche se in realtà questo significasse "saltellare" in tutto il mondo alla ricerca del server corretto.

Inizialmente era un semplice processo a senso unico: si eseguiva una richiesta a un server, ottenendo un file, che il software di visualizzazione del computer



client interpretava, formattandolo sul sistema locale. Ben presto, tuttavia, gli utenti hanno iniziato a pretendere qualcosa di più che salvare pagine provenienti da un server: hanno cominciato a voler restituire informazioni ai server, consultarne i database, aggiungere informazioni o inoltrare ordini, il che, peraltro, ha richiesto l'adozione di particolari misure di sicurezza. Questi sono i cambiamenti che hanno contraddistinto lo sviluppo del Web.

I browser hanno permesso di compiere un notevole passo in avanti, introducendo il concetto che un documento possa essere visualizzato indifferentemente su qualsiasi tipo di computer. I primi browser erano strumenti molto primitivi, che avevano tendenza a bloccarsi rapidamente: poco interattivi, spesso causavano la congestione sia del server sia della rete Internet, poiché ogni volta che era necessario eseguire un'attività che richiedesse programmazione dovevano inviare informazioni da elaborare al server; potevano trascorrere molti secondi o minuti prima di scoprire che qualcosa, nella richiesta, era stato trasmesso in modo errato. Poiché il browser era solo uno spettatore, non era in grado di eseguire neppure le elaborazioni più semplici: d'altra parte era uno strumento sicuro, perché incapace di eseguire sul sistema locale qualsiasi programma che potesse contenere bug o virus.

Per risolvere questo problema si sono adottati vari approcci. In primo luogo sono stati perfezionati gli standard grafici, per consentire che i browser visualizzassero animazioni e video. Il resto del problema è stato risolto introducendo la capacità di eseguire programmi lato client, nel browser: un approccio che è conosciuto come *programmazione lato client*.

Programmazione lato client

Lo schema iniziale server-browser del Web forniva contenuto interattivo, ma l'interattività era completamente a carico del server. Il server produceva pagine statiche per il browser client, che si limitava a interpretarle e visualizzarle. Il linguaggio HTML (*HyperText Markup Language*) dispone di semplici meccanismi per la raccolta dei dati: caselle di testo, caselle di controllo, pulsanti radio, elenchi ed elenchi a discesa, e un pulsante che può essere programmato unicamente per reimpostare i dati del modulo o per inviare i dati del modulo al server. Questa trasmissione passa attraverso la cosiddetta interfaccia CGI (*Common Gateway Interface*), presente in tutti i server web: è lo stesso testo contenuto nei dati inviati che indica alla CGI come trattarlo.

L'azione più comune consiste nell'eseguire un programma presente sul server, in una directory chiamata generalmente *cgi-bin*: in effetti, se osservate la casella degli indirizzi nella parte superiore del vostro browser, quando fate clic su un pulsante in una pagina web spesso noterete la stringa *cgi-bin* al-



l'interno dell'indirizzo. Questi programmi possono essere scritti, in pratica, in qualsiasi linguaggio: Perl è una scelta ricorrente poiché progettato per manipolare e interpretare il testo, quindi installabile su qualsiasi server a prescindere dal processore e dal sistema operativo, tuttavia in tempi recenti è emerso il linguaggio Python (www.python.org), caratterizzato da maggiore potenza e semplicità.

Molti tra i siti web più potenti oggi si basano rigorosamente sulle CGI che consentono di eseguire pressoché qualsiasi operazione. Tuttavia i siti web basati su CGI possono ben presto diventare eccessivamente complessi da gestire, oltre a presentare il problema dei tempi di risposta, che dipendono dalla quantità di dati da inviare e dal carico del server e di Internet. In aggiunta, l'avvio di un programma CGI è generalmente lento. I primi progettisti del Web non avevano previsto quanto rapidamente la banda disponibile si sarebbe esaurita a causa del tipo di applicazioni che sarebbero state sviluppate.

Per esempio, è quasi impossibile visualizzare grafici dinamici in modo fluido, perché per ogni versione del grafico occorre creare un file GIF (*Graphics Interchange Format*) e spostarlo dal server al client. Inoltre, tutti voi avrete certamente sperimentato il processo di convalida dei dati per un modulo di immissione web: fate clic sul tasto Invio; i dati vengono spediti al server, il quale avvia un programma CGI che rileva un errore, e formatta una pagina HTML per informarvi di esso, che poi vi invia; a quel punto dovete nuovamente tornare alla pagina precedente e provare di nuovo. Un procedimento non soltanto lento, ma anche inelegante.

La soluzione è rappresentata dalla programmazione lato client. Molti computer desktop su cui sono attivi i browser web sono veri e propri motori potenti, in grado di eseguire una grande mole di lavoro: con l'HTML statico, invece, si limitano a rimanere in attesa che il server invii la pagina prevista. Il termine *programmazione lato client* indica che il browser web viene sfruttato per ogni possibile lavoro; il risultato per l'utente è un'attività più rapida e interattiva con il sito.

Il problema delle discussioni relative alla programmazione lato client è che esse non sono molto diverse da quelle di programmazione generica. I parametri sono quasi gli stessi, mentre cambia la piattaforma: un browser web è una sorta di sistema operativo ridotto. In definitiva si tratta sempre di programmare, e questo giustifica in qualche modo la miriade di problemi e soluzioni prodotti dalla programmazione lato client. La parte successiva di questo paragrafo fornisce una panoramica degli argomenti e degli approcci della programmazione lato client.



Plug-in

Una delle evoluzioni più interessanti nella programmazione lato client è costituita dallo sviluppo dei plug-in, un meccanismo mediante il quale il programmatore aggiunge nuove funzionalità al browser tramite il download di una porzione di codice che si collega opportunamente al browser. In pratica è come se si dicesse al browser che, a partire da un certo momento, esso sarà in grado di eseguire una nuova attività: ovviamente il plug-in deve essere scaricato una sola volta. I plug-in permettono di aggiungere ai browser funzionalità veloci e potenti, tuttavia scrivere plug-in è un'operazione tutt'altro che banale, e soprattutto non è un compito che si desidera svolgere nel processo di realizzazione di un sito web. Dal punto di vista della programmazione lato client il valore di un plug-in risiede nel consentire al programmatore esperto di sviluppare estensioni, che possono essere integrate nel browser senza l'autorizzazione del produttore del browser stesso. In questo senso il plug-in fornisce una "backdoor" che consente la creazione di nuovi linguaggi di programmazione lato client, benché non tutti questi linguaggi siano implementati come plug-in.

Linguaggi di scripting

I plug-in hanno permesso lo sviluppo dei linguaggi di scripting "lato browser". Con un linguaggio di scripting il codice sorgente del programma lato client è inserito direttamente nella pagina HTML, e il plug-in che lo interpreta si attiva in modo automatico durante la visualizzazione della pagina. I linguaggi di scripting sono generalmente facili da comprendere e, poiché sono semplici inserti testuali che fanno parte di una pagina HTML, vengono caricati molto rapidamente, come parte della richiesta inviata al server per fornire la pagina.

L'inconveniente è che il codice è "esposto", e può essere visto (e potenzialmente rubato) da chiunque. Tuttavia, in genere i linguaggi di scripting non eseguono operazioni molto ricercate, pertanto questo aspetto assume un'importanza relativa.

Un linguaggio di scripting normalmente supportato dai browser web *privi* di plug-in è JavaScript: questo linguaggio ha soltanto una vaga somiglianza con Java, e il suo utilizzo richiede una curva di apprendimento supplementare. In effetti, JavaScript è stato chiamato in questo modo solamente per "cavalcare l'onda" del marketing Java. Purtroppo, in origine quasi tutti i browser web hanno implementato JavaScript in modi diversi tra loro, e in qualche caso anche dalle varie versioni dei propri JavaScript. La standardizzazione di JavaScript come *ECMAScript* è stata un eccellente contributo, tuttavia la sua diffusione presso i diversi browser ha richiesto molto tempo: di contro



non ha aiutato Microsoft, che promuoveva invece VBScript, anch'esso caratterizzato da vaghe somiglianze con JavaScript.

In generale, per realizzare un programma il cui codice sia eseguibile in tutti i browser è necessario programmare in una forma "simil-JavaScript". La gestione degli errori e la correzione del codice JavaScript possono essere descritte soltanto come un vero e proprio incubo: come prova di questa difficoltà, in tempi recenti soltanto Google (per GMail) è riuscita a creare codice JavaScript veramente complesso, e ciò ha richiesto impegno e competenze notevoli.

Questo dimostra che i linguaggi di scripting usati all'interno dei browser web, di fatto, sono progettati per risolvere tipi specifici di problemi, in particolare la creazione di interfacce utente grafiche, o GUI (*Graphical User Interface*), più ricche e interattive. Tuttavia, un linguaggio di scripting potrebbe risolvere l'80% dei problemi che si riscontrano nella programmazione lato client. È possibile che i vostri problemi di programmazione rientrino in questa percentuale: in tal caso, dal momento che i linguaggi di scripting possono consentire uno sviluppo più facile e veloce, dovreste valutare l'utilizzo di un linguaggio di scripting prima di esaminare soluzioni più complesse, quali la programmazione Java.

Java

Se un linguaggio di scripting può risolvere l'80% dei problemi di programmazione lato client, che dire del 20% rimanente, lo "zoccolo duro"? In questi casi, Java è una soluzione diffusa: non solo è un linguaggio di programmazione potente, concepito per essere sicuro, multipiattaforma e internazionale, ma viene anche esteso di continuo per offrire funzionalità linguistiche e librerie che gestiscono in modo elegante problemi che è difficile gestire con i linguaggi di programmazione tradizionali, come la concorrenza, l'accesso ai database, la programmazione e l'elaborazione distribuita in rete. Java permette la programmazione lato client tramite gli *applet* e con *Java Web Start*.

Un applet è un mini-programma che viene mandato in esecuzione soltanto dall'interno di un browser web. L'applet viene scaricato automaticamente come parte di una pagina web, nello stesso modo in cui, per esempio, viene scaricato un file grafico. Quando l'applet è attivato esegue un programma. Questa è parte della sua attrattiva: fornisce al server un modo per distribuire automaticamente il software client, nel momento in cui l'utente ne ha bisogno, non prima. L'utente ottiene l'ultima versione del software client, senza errori di sorta né complesse reinstallazioni.



Grazie al modo in cui Java è progettato, al programmatore basta creare un solo programma, che funzionerà automaticamente su tutti i computer dotati di browser con un interprete Java incorporato, ovvero la grande maggioranza dei sistemi. Poiché Java è un linguaggio di programmazione completo, vi consente di eseguire quanto più lavoro possibile sul client, prima e dopo aver eseguito le richieste al server. Per esempio, non sarete costretti a trasmettere un modulo via Internet per scoprire che una data o un altro parametro sono errati, e il computer client potrà creare rapidamente un grafico dei dati, invece di aspettare che il server crei un'immagine e la trasmetta al client. Non solo beneficerete immediatamente di velocità e prontezza nelle risposte, ma il traffico di rete e il carico sui server risulteranno ridotti, impedendo all'intera rete Internet di rallentare oltremisura.

Alternative

In verità, gli applet Java non hanno avuto una vita molto lunga dopo la gloria iniziale. All'apparire di Java il mondo della programmazione si era dimostrato entusiasta degli applet, perché questi avrebbero consentito una seria programmabilità lato client, incrementando i tempi di risposta e riducendo i requisiti di ampiezza di banda necessari per le applicazioni basate su Internet. Sono state intraviste vaste possibilità.

Effettivamente sul Web si possono trovare alcuni applet molto brillanti, tuttavia un vero e proprio "esodo" a favore degli applet non si è mai verificato. Il problema principale è stato forse che il trasferimento dei 10 Mb necessari per installare il JRE (*Java Runtime Environment*) era una prospettiva sproporzionata per l'utente medio; il fatto che Microsoft abbia scelto di non includere JRE con Internet Explorer, inoltre, probabilmente ne ha deciso il destino. Comunque sia, gli applet Java non sono mai stati usati su vasta scala.

Malgrado ciò, gli applet e le applicazioni *Java Web Start* sono ancora utili in alcune situazioni. In qualsiasi momento abbiate il controllo della macchina dell'utente, per esempio nell'ambito di un'azienda, questa tecnologia si rivela adatta per distribuire e aggiornare le applicazioni client, risparmiando considerevoli quantità di tempo, sforzi e denaro, in particolare laddove siano necessari aggiornamenti frequenti.

Nel Volume 3, Capitolo 2, "Interfacce grafiche (GUI)" esaminerete Flex di Macromedia, una nuova tecnologia promettente che consente di creare applicazioni simili agli applet, basati su Flash. Tenuto conto che Flash Player è disponibile per circa il 98% dei browser web, inclusi quelli per Windows, Linux e Mac, può ormai essere considerato uno standard; in aggiunta, l'installazione e l'aggiornamento di Flash Player sono operazioni rapide e facili. Il linguaggio ActionScript è basato su ECMAScript, pertanto ragionevol-



mente familiare, tuttavia Flex permette di programmare senza preoccuparsi delle specifiche del browser, risultando più interessante di JavaScript: per la programmazione lato client, questa è un'alternativa meritevole di essere presa in considerazione.

.NET e C#

Per un certo periodo il principale antagonista degli applet Java è stato la tecnologia ActiveX di Microsoft, seppure supportata soltanto dai client in ambiente Windows. Da allora Microsoft ha realizzato un completo concorrente di Java, rappresentato dall'ambiente .NET e dal linguaggio di programmazione C#. La piattaforma .NET è del tutto simile alle librerie Java e alla JVM (*Java Virtual Machine*), la piattaforma software sulla quale vengono eseguiti i programmi Java, e il linguaggio C# mostra sorprendenti analogie con Java.

Ne è risultato senza dubbio il miglior lavoro che Microsoft abbia svolto nell'ambito dei linguaggi di programmazione e degli ambienti di sviluppo. Naturalmente, Microsoft ha avuto il considerevole vantaggio di analizzare pregi e difetti del funzionamento di Java, e ha potuto basarsi su questo linguaggio. In ogni caso, va preso atto che per la prima volta Java ha un reale concorrente: di conseguenza, i progettisti Java di Sun hanno esaminato a fondo C# e i motivi che potrebbero indurre i programmati a desiderare di utilizzare questo linguaggio, e hanno risposto con Java SE5, che apporta miglioramenti significativi a Java.

Attualmente la domanda più importante relativa a .NET, che costituisce anche il principale punto debole, è se Microsoft ne permetterà la portabilità completa su altre piattaforme. Sembra che non vi siano problemi a riguardo, e il progetto Mono (www.go-mono.com) offre un'implementazione parziale di .NET per i sistemi Linux: tuttavia, finché tale implementazione non sarà completa e fino a quando Microsoft non avrà deciso se ostacolare o meno questo progetto, l'utilizzo di .NET come soluzione multi-piattaforma è ancora una scommessa rischiosa.

Confronto tra Internet e intranet

Il Web è la soluzione più generica alla questione client/server, in quanto rende ragionevole l'utilizzo della stessa tecnologia per risolvere un sottoinsieme del problema, in particolare quello classico dell'impiego del paradigma client/server all'interno di un'azienda. Nell'approccio client/server tradizionale si presenta sempre la tematica di dover far coesistere diversi tipi di computer client, in aggiunta alle difficoltà di installare nuovo software client, problemi risolti agevolmente grazie al browser web e alla programmazione lato client.



Si definisce *intranet* l'uso della tecnologia web per realizzare una rete informativa limitata a una particolare azienda. Le reti intranet sono caratterizzate da una maggiore sicurezza rispetto a Internet, poiché consentono di controllare fisicamente l'accesso ai server interni dell'azienda. Per quanto concerne l'addestramento del personale, una volta compresi i concetti generali dei browser risulta generalmente più semplice occuparsi delle differenze relative alle pagine e agli applet, pertanto la curva di apprendimento sembra sia ridotta.

Il problema della sicurezza porta a una delle divisioni che sembrano formarsi spontaneamente nel mondo della programmazione lato client. Se il vostro programma è in esecuzione su Internet non potete sapere su quale piattaforma verrà eseguito, pertanto dovete prestare ancor più attenzione a non distribuire codice contenente bug: vi occorre quindi uno strumento multiplattforma e sicuro, quale può essere un linguaggio di scripting o Java.

Se il programma opera su un'intranet potreste essere soggetti a vincoli di vario tipo. Considerate che non è affatto raro che i computer siano tutti in architettura Intel/Windows. Su un'intranet siete interamente responsabili della qualità del codice ma potete risolvere i bug non appena vengono rilevati. Inoltre, potreste già avere a disposizione vecchio codice usato in soluzioni client/server più tradizionali, mediante il quale installare fisicamente il software client ogni volta che si renda necessario un aggiornamento.

Il tempo richiesto dall'installazione degli aggiornamenti è il motivo che più di ogni altro incoraggia a passare ai browser, nei quali gli aggiornamenti sono invisibili e automatici (Java Web Start risolve anche questo problema). Se state lavorando a una rete intranet, l'orientamento più sensato da adottare è la soluzione più breve che vi permetta di riutilizzare la base di codice preesistente, anziché codificare nuovamente i vostri programmi in un altro linguaggio.

Di fronte a questo sorprendente insieme di soluzioni al problema della programmazione lato client, il miglior piano d'attacco è un'attenta analisi costi-benefici. Considerate i vincoli del vostro problema e quale potrebbe essere la soluzione più diretta. Poiché la programmazione lato client è ancora in fase di sviluppo, è sempre preferibile scegliere l'approccio di sviluppo più rapido per la situazione specifica. Questo è un atteggiamento aggressivo per prepararsi allo scontro con gli inevitabili problemi di sviluppo del software.



Programmazione lato server

Questa presentazione ha ignorato finora l'argomento della programmazione lato server, che è probabilmente il settore in cui Java ha riscosso maggiore successo. Che cosa accade quando si esegue una richiesta a un server? Nella maggior parte dei casi si tratta della semplice richiesta di invio di un file, che il browser ha poi cura di interpretare nel modo opportuno: come pagina HTML, immagine grafica, applet Java, script e così via.

Richieste più complesse a un server generalmente sottintendono transazioni di database. Una situazione comune è quella di una richiesta per una complessa ricerca su un database, a fronte della quale il server formatta i risultati in una pagina HTML che provvede poi a trasmettere. Naturalmente, se il sistema client ha una maggiore "intelligenza elaborativa" fornитagli da Java o da un linguaggio di scripting, possono essere inviati i dati grezzi che verranno formattati direttamente sul client, in modo più rapido e con onere inferiore per il server.

Un altro tipo di richiesta prevede la registrazione di dati in un database, per esempio quando si aderisce a un newsgroup o si inoltra un ordine: questa operazione richiederà l'esecuzione di modifiche sul database. Tali richieste al database devono essere elaborate per mezzo di codice eseguito sul server, procedimento noto come *programmazione lato server*.

Tradizionalmente la programmazione lato server veniva svolta utilizzando Perl, Python, C++ o altri linguaggi idonei alla creazione di programmi CGI; in seguito sono comparse tecnologie più sofisticate, tra cui i server web basati su Java che permettono di eseguire tutta la programmazione lato server in Java, ricorrendo ai cosiddetti *servlet*.

I servlet e i loro discendenti, le JSP (*Java Server Pages*), sono due dei motivi principali che inducono gli sviluppatori di siti web a passare a Java, soprattutto perché eliminano il problema di gestire browser con funzionalità diverse. La programmazione lato server è trattata in dettaglio nel volume *Thinking in Enterprise Java*, disponibile su www.mindview.net.

A prescindere da tutto quanto si è detto finora riguardo a Java su Internet, rimane il fatto che è un linguaggio di programmazione di uso generale capace di risolvere problemi risolvibili anche con altri linguaggi. La forza di Java non consiste unicamente nella portabilità, ma anche nella programmabilità e robustezza, nella sua ricca libreria standard e in quelle fornite da terze parti e nelle nuove librerie che continuano a essere sviluppate.



Riepilogo

Sapete che la programmazione procedurale è composta dalla definizione dei dati e dalle chiamate a funzioni. Per comprendere il significato di un programma di questo tipo occorre lavorarci, esplorare le chiamate alle funzioni e i concetti di basso livello, fino a creare un modello mentale. Nella progettazione dei programmi procedurali, quindi, è necessario ricorrere a rappresentazioni intermedie: per loro natura, questi programmi tendono a essere fuorvianti poiché maggiormente orientati al sistema anziché al problema da risolvere.

Dal momento che la programmazione orientata agli oggetti integra nuovi concetti a quelli che già contraddistinguono un linguaggio procedurale, l'impressione immediata di molti lettori è che un programma Java possa essere più complesso del suo corrispondente procedurale. In questo senso rimarrete piacevolmente sorpresi: di solito, un programma Java ben scritto è molto più semplice e facile da capire di un programma procedurale equivalente. Ciò che vedrete sono le definizioni degli oggetti che rappresentano i concetti nello spazio del problema (invece degli argomenti della rappresentazione nel sistema) e i messaggi inviati a tali oggetti per rappresentare le attività in questo spazio. Una delle caratteristiche più apprezzate della programmazione a oggetti è la facilità di comprensione del codice di un programma ben progettato, semplicemente leggendolo. Di norma, anche la quantità di codice è ridotta, poiché molti problemi vengono risolti riutilizzando il codice esistente nelle librerie.

Il linguaggio Java e la programmazione OOP potrebbero non essere adatti a tutti gli sviluppatori. È importante che valutiate i vostri bisogni per determinare se Java sarà in grado di soddisfarli in modo ottimale, o se potrete appagare meglio queste necessità privilegiando un ambiente di programmazione alternativo, incluso quello che già state utilizzando. Qualora le vostre esigenze per il prevedibile futuro siano molto specializzate, e se avete vincoli specifici, Java potrebbe non essere la scelta appropriata; in tal caso considerate eventuali alternative: in particolare l'autore raccomanda di valutare il linguaggio Python (www.python.org). Dopo queste considerazioni, se avrete optato per Java come linguaggio di sviluppo, avrete quantomeno compreso quali sono le opzioni e disporrete di una visione chiara delle ragioni che hanno motivato la vostra decisione.

Tutto è un oggetto

"Se parliamo una lingua diversa, in qualche modo percepiamo un mondo diverso."

Ludwig Josef Johann Wittgenstein (1889-1951)



Malgrado sia basato su C++, Java è un linguaggio orientato agli oggetti più "puro".

Sia C++ sia Java sono linguaggi ibridi, tuttavia per Java i progettisti hanno ritenuto che l'ibridazione non fosse così importante quanto lo era in C++. Un linguaggio ibrido permette di usare stili di programmazione diversi; la ragione per cui C++ è ibrido è la necessità di renderlo retrocompatibile con il linguaggio C.

Poiché C++ è un "sovrainsieme" del linguaggio C stesso, include molte caratteristiche indesiderabili di quest'ultimo, che possono rendere alcuni aspetti del C++ oltremodo complessi.

Il linguaggio Java presuppone che vogliate programmare soltanto a oggetti. Ciò implica che, prima di iniziare il lavoro, dovete adattare la vostra mente a un mondo orientato agli oggetti.

Il vantaggio di questo sforzo iniziale è la capacità di programmare in un linguaggio più semplice da apprendere e utilizzare rispetto ai molti altri linguaggi OOP. In questo capitolo esaminerete i componenti di base di un programma Java e vedrete che (quasi) tutto in Java è un oggetto.



Gli oggetti si manipolano per riferimento

Ogni linguaggio di programmazione ha metodi propri per gestire gli elementi in memoria. Il programmatore deve essere costantemente consapevole del tipo di manipolazione che sta eseguendo. State manipolando l'elemento direttamente oppure trattando con una sorta di rappresentazione indiretta, per esempio un puntatore in C/C++, che richiede una sintassi speciale?

Tutto ciò, in Java, è stato semplificato: tutto viene considerato come un oggetto, utilizzando un'unica sintassi coerente. Malgrado questo, in realtà l'identificativo da manipolare è un "riferimento" a un oggetto.¹

Potete assimilare questo concetto a un televisore (l'oggetto) e al suo telecomando (il riferimento). Fino a quando conservate questo riferimento siete collegati all'oggetto televisore; tuttavia, se decidete di "cambiare canale" o di "diminuire il volume", vi servite del riferimento, che a sua volta interviene sull'oggetto. Se volete passeggiare nella stanza continuando a controllare il televisore prenderete con voi il telecomando/riferimento, non il televisore.

Inoltre, il telecomando può esistere per proprio conto anche senza televisore. Il fatto che esista un riferimento non implica necessariamente che esista un oggetto a esso collegato. Così, se volete conservare una parola o una frase, create un riferimento a un oggetto stringa (**String**):

```
String s;
```

In tal modo, però, avete creato soltanto il riferimento, non l'oggetto. A questo punto, se cercaste di inviare un messaggio a **s** otterreste un errore: **s** in

1. Questo è un punto controverso. Alcuni lettori potrebbero obiettare che "si tratta chiaramente di un puntatore", tuttavia ciò presupporrebbe un'implementazione di fondo. Inoltre, la sintassi dei riferimenti Java li rende più simili a riferimenti C++ che a puntatori di questo linguaggio. Nella prima edizione di questo libro l'autore aveva deciso di ricorrere a un nuovo termine, *handle*, dal momento che tra i riferimenti C++ e quelli Java esistono notevoli differenze. L'autore proveniva da un'esperienza di C++ e voleva evitare di confondere i programmatore di questo linguaggio, che (così riteneva all'epoca) avrebbero rappresentato la maggioranza dei lettori. Nella seconda edizione l'autore ha poi ritenuto che il termine "riferimento" (*reference*) fosse sicuramente più corrente e che i lettori provenienti dal C++ avrebbero avuto minori difficoltà con la terminologia legata ai riferimenti. Tuttavia, alcuni tecnici disapprovano anche l'impiego del termine "riferimento". Essi sostengono che "è sbagliato affermare che Java supporta il passaggio per riferimento", poiché gli identificativi degli oggetti Java sono, in realtà, "riferimenti agli oggetti". Pertanto il passaggio non avviene per riferimento, ma tutto viene "passato per valore": anzi, più correttamente, "passando il riferimento a un oggetto per valore". Certo, si potrebbe avere qualcosa da obiettare sulla scrupolosità di tali spiegazioni complesse, in ogni caso l'autore ritiene che il suo approccio semplifichi la comprensione del concetto senza urtare la suscettibilità di nessuno.



effetti non è collegato a nulla, poiché non esiste nessun televisore. In questo senso, una tecnica più pratica consiste nell'inizializzare sempre un riferimento al momento della sua creazione:

```
String s = "asdf";
```

Questo esempio evidenzia già una caratteristica speciale di Java: la possibilità di inizializzare direttamente le stringhe fornendo testo tra virgolette. Di norma, per gli oggetti, è necessario ricorrere a un tipo di inizializzazione più generico.

Tutti gli oggetti devono essere creati

Quando create un riferimento volete anche che sia collegato al nuovo oggetto: tale operazione di solito viene eseguita con l'operatore **new**. La parola chiave **new** esprime il concetto: "crea un nuovo oggetto di questo tipo". Pertanto, a proposito dell'esempio precedente potreste scrivere:

```
String s = new String("asdf");
```

Questa stringa di codice significa non soltanto "crea un nuovo oggetto **String**", ma spiega anche come generare la stringa, fornendo una serie di caratteri iniziale.

Naturalmente, oltre a **String** Java dispone di molti altri tipi pronti all'uso, ma ciò che è più rilevante è la possibilità di creare i vostri tipi personalizzati. Infatti, la creazione di nuovi tipi è l'attività primaria della programmazione Java, ed è ciò che imparerete nel resto di questo volume.

La registrazione dei dati

Ora è utile illustrare alcuni aspetti di ciò che si verifica quando il programma è in esecuzione: in particolare come è organizzata la memoria. Le aree in cui è possibile memorizzare i dati sono cinque.

- Registri.** Questo è il meccanismo di registrazione più veloce, poiché si trova in una zona particolare: all'interno del processore. Essendo in numero molto limitato, tuttavia, i registri vengono allocati solo quando non sia possibile fare diversamente. Non è permesso controllare direttamente i registri, né desumerne l'esistenza da qualche elemento presente nei programmi. I linguaggi C e C++, invece, consentono di richiedere al compilatore l'allocazione dei registri.



2. **Lo stack.** Generalmente collocato nella memoria RAM (*Random Access Memory*, memoria ad accesso casuale), è tuttavia collegato direttamente al processore tramite il *puntatore allo stack* (*stack pointer*): tale puntatore viene spostato in basso per creare nuova memoria e verso l'alto per rilasciarla. Questo è un modo veramente rapido ed efficiente per allocare memoria, secondo solo ai registri del processore. Durante la compilazione il compilatore Java deve conoscere la durata esatta di quanto deve essere registrato nello stack, e questo pone limiti alla flessibilità dei programmi: di conseguenza in quest'area Java memorizza, in particolare, i riferimenti agli oggetti, mentre gli oggetti veri e propri sono conservati altrove.
3. **La memoria heap.** Anch'essa collocata in RAM, è la memoria di uso generale nella quale vengono conservati tutti gli oggetti Java. La caratteristica interessante della memoria heap è che, a differenza dello stack, il compilatore non ha bisogno di sapere per quanto tempo i dati devono essere conservati: per questo motivo la memorizzazione nella memoria heap risulta particolarmente flessibile. Ogni volta che avete bisogno di un oggetto non dovete fare altro che scrivere il codice per crearlo, usando l'operatore **new**, e lo spazio sarà allocato nell'heap al momento di eseguire il codice. L'inconveniente che controbilancia tanta flessibilità è dato dal maggior tempo che sarebbe necessario per allocare e riordinare la memoria heap, rispetto allo stack, se anche in Java fosse possibile creare oggetti nello stack, come avviene in C++.
4. **Memorizzazione statica.** Di norma i valori costanti vengono posizionati direttamente nel codice del programma, e sono sicuri in quanto il codice non cambia. Talvolta le costanti sono “segregate”, in modo da poter essere eventualmente registrate nella memoria ROM (*Read Only Memory*, memoria di sola lettura) dei “sistemi incapsulati” (*embedded system*).²
5. **Memorizzazione extra-RAM.** Se i dati “vivono” completamente al di fuori di un programma possono esistere anche mentre il programma non è in esecuzione, cioè fuori dal suo controllo. I due esempi principali sono gli *oggetti streamed*, ovvero quelli che vengono trasformati in flussi di byte, di norma da trasmettere a un altro sistema, e gli *oggetti persistenti*, registrati su disco affinché conservino il proprio stato anche quando il programma è terminato. La tecnica consiste nel trasformare gli oggetti in una forma che possa esistere su altri supporti, poi ripristinabile in normali oggetti di memoria RAM quando sia necessario. Java fornisce supporto a un tipo di

2. Un esempio di questa tecnica è il cosiddetto *pool* di stringhe (*string pool*): tutte le stringhe letterali e le espressioni costanti con valore **String** vengono automaticamente confinate in un'area statica speciale.



“persistenza leggera” (*lightweight persistence*), mentre framework come JDBC e Hibernate offrono un supporto più elaborato per conservare e recuperare dai database le informazioni sugli oggetti.

Un caso speciale: i tipi primitivi

Nella programmazione vi servirete spesso di alcuni tipi che godono di un trattamento riservato: potete considerarli come una sorta di tipi “primitivi”. Il motivo di questo trattamento speciale risiede nel fatto che la creazione di un oggetto con **new**, specialmente se si tratta di una semplice variabile, non è considerata un'operazione particolarmente efficiente, poiché **new** registra i nuovi oggetti nell'heap. Per questi tipi Java adotta l'approccio ereditato dal C/C++: in pratica, invece di creare la variabile usando **new**, genera una variabile “automatica” che *non è un riferimento*. La variabile contiene direttamente il valore e viene memorizzata nello stack, quindi è molto più efficiente.

Java determina le dimensioni di ogni tipo primitivo, che, a differenza di quanto accade nella maggior parte dei linguaggi, rimangono invariate passando da un'architettura hardware a un'altra. Queste dimensioni invariabili sono uno dei motivi che rendono un programma Java maggiormente portabile rispetto ai programmi scritti in quasi tutti gli altri linguaggi.

Tipo primitivo	Dimensioni	Minimo	Massimo	Tipo wrapper
boolean	—	—	—	Boolean
char	16 bit	Unicode 0	Unicode $2^{16}-1$	Character
byte	8 bit	-128	+127	Byte
short	16 bit	-2^{15}	$+2^{15}-1$	Short
int	32 bit	-2^{31}	$+2^{31}-1$	Integer
long	64 bit	-2^{63}	$+2^{63}-1$	Long
float	32 bit	IEEE754 (*)	IEEE754 (*)	Float
double	64 bit	IEEE754 (*)	IEEE754 (*)	Double
void	—	—	—	Void

(*) Troverete maggiori informazioni sullo standard IEEE per il calcolo in virgola mobile nella pagina web all'indirizzo http://it.wikipedia.org/wiki/IEEE_754.

Tutti i tipi numerici sono con segno (*signed*), quindi non aspettatevi tipi privi di segno.



Le dimensioni del tipo **boolean** non sono specificate; esso è definito soltanto come in grado di assumere i valori letterali **true** o **false**.

Le classi “wrapper” per i tipi di dato primitivi permettono di creare un oggetto non-primitivo sull’heap, allo scopo di rappresentare il tipo primitivo corrispondente. Per esempio:

```
char c = 'x';
Character ch = new Character(c);
```

In alternativa potreste usare anche:

```
Character ch = new Character('x');
```

La funzionalità *autoboxing* di Java SE5 si incarica di convertire automaticamente un tipo primitivo in un tipo wrapper:

```
Character ch = 'x';
```

e viceversa:

```
char c = ch;
```

Numeri a precisione elevata

Java include due classi per l’esecuzione di calcoli ad alta precisione: **BigInteger** e **BigDecimal**. Benché queste classi abbiano numerose analogie con le classi wrapper, non hanno un tipo primitivo corrispondente.

Entrambe le classi dispongono di metodi che consentono operazioni analoghe a quelle eseguibili sui tipi primitivi. In pratica, un **BigInteger** o un **BigDecimal** permettono di eseguire le stesse operazioni possibili con un **int** o un **float**: dovete semplicemente avere l’accortezza di eseguire tali operazioni mediante chiamate a metodi anziché ricorrere agli operatori. Inoltre, poiché implicano una maggiore attività, le operazioni saranno più lente: al solito, si tratta di trovare il compromesso migliore tra velocità e precisione.

BigInteger supporta numeri interi di precisione arbitraria. Potete quindi rappresentare con accuratezza valori interi di qualsiasi dimensione senza perdere alcuna informazione durante le operazioni.

BigDecimal, invece, si utilizza per i numeri a virgola fissa con precisione arbitraria, usati per esempio nei calcoli finanziari.



Per ulteriori dettagli sui costruttori e sui metodi richiamabili per queste due classi consultate la documentazione JDK.

Gli array in Java

In un modo o nell’altro, tutti i linguaggi di programmazione supportano gli array. Utilizzarli in C/C++ è considerato rischioso, poiché gli array sono semplici blocchi di memoria: se un programma accede all’array al di fuori del suo blocco di memoria o usa la memoria prima dell’inizializzazione (due errori di programmazione piuttosto comuni), ne deriveranno risultati imprevedibili.

Uno degli obiettivi principali di Java è la sicurezza; per questo molti dei problemi che affliggono i programmati in C/C++ non si riscontrano in Java. Un array in Java viene sempre inizializzato e non è accessibile al di fuori del suo ambito. Il controllo dell’ambito comporta un costo gestionale, in termini di una piccola quantità di memoria aggiuntiva su ogni array e della verifica dell’indice al momento dell’esecuzione: la tesi sostenuta, tuttavia, è che una maggiore sicurezza e produttività valgano questo prezzo. Occorre considerare, inoltre, che talvolta Java riesce a ottimizzare queste operazioni.

Quando create un array di oggetti, in realtà generate un array di riferimenti, ognuno dei quali viene automaticamente inizializzato a un valore speciale, indicato con la propria parola chiave: **null**. Quando Java incontra la parola chiave **null** riconosce che il riferimento in questione non punta a nessun oggetto. Dovete assegnare un oggetto a ogni riferimento prima di utilizzarlo; se tentate di usare un riferimento che ha ancora valore **null**, il problema verrà segnalato in fase di esecuzione. In tal modo Java permette di evitare gli errori tipici degli array.

Potete anche creare un array di tipi primitivi: anche in questo caso il compilatore garantisce l’inizializzazione, poiché azzerà la memoria per l’array in uso.

Gli array verranno trattati in dettaglio nei capitoli successivi.

Non occorre mai distruggere un oggetto

Nella maggior parte dei linguaggi il concetto di ciclo di vita di una variabile è un elemento importante dell’attività di programmazione. Quanto tempo dura la variabile? Supponendo di dover procedere alla sua distruzione, quando si dovrebbe farlo? La confusione sulla durata delle variabili può produrre

molti errori: in questo paragrafo vedrete come Java semplifica l'argomento, gestendo l'attività di cleanup per vostro conto.

Ambito di visibilità

Molti linguaggi procedurali hanno il concetto di ambito (*scope*), che determina sia la visibilità sia la durata dei nomi definiti all'interno dell'ambito. In C, C++ e Java, l'ambito viene indicato da un coppia di parentesi graffe ({}). Per esempio:

```
{
    int x = 12;
    // È disponibile soltanto x
    {
        int q = 96;
        // Sono disponibili sia x sia q
    }
    // È disponibile soltanto x,
    // q è "fuori ambito" (out of scope)
}
```

Una variabile definita all'interno di un ambito è disponibile solo per la durata di quell'ambito.³

Non potete invece eseguire questa operazione, sebbene sia ammessa in C e C++:

```
{
    int x = 12;
    {
        int x = 96; // Illegale
    }
}
```

³ Qualsiasi testo presente dopo un //, fino alla fine della riga, è considerato un commento. L'indentazione rende il codice più facilmente leggibile: poiché Java è un linguaggio in formato libero, spazi supplementari, tabulazioni e carriage return non influiscono sul programma risultante.

Il compilatore segnalerà che la variabile x è già stata definita. La capacità, tipica di C e C++, di “nascondere” una variabile in un ambito più ampio non è presente, poiché i progettisti di Java hanno ritenuto che rendesse confusa la programmazione.

Ambito di visibilità degli oggetti

Gli oggetti Java non hanno lo stesso ambito dei tipi primitivi. Quando create un oggetto Java con l'operatore new, l'oggetto rimane in essere anche oltre la fine dell'ambito. Pertanto, se utilizzate

```
{
    String s = new String("a string");
} // Fine dell'ambito
```

il riferimento s scomparirà alla fine dell'ambito, tuttavia l'oggetto String cui faceva riferimento s occupa ancora memoria. In questo frammento di codice non vi è alcun modo per accedere all'oggetto oltre l'ambito, dal momento che l'unico riferimento a esso è esterno all'ambito stesso. Nei capitoli che seguono vedrete che il riferimento all'oggetto può essere passato e duplicato nel corso del programma.

È evidente che, poiché gli oggetti creati con l'operatore new esistono finché lo si ritiene opportuno, in Java moltissimi problemi di programmazione tipici di C++ semplicemente “svaniscono”. In C++ dovete non solo assicurarvi che gli oggetti esistano finché ne avete bisogno, ma anche ricordarvi di distruggerli quando non vi occorrono più.

A questo punto è d'obbligo un'altra domanda: se Java lascia sopravvivere gli oggetti, che cosa impedisce loro di riempire la memoria e bloccare il vostro programma? Questo è esattamente il tipo di problema che si verificherebbe in C++. Qui entra in gioco un po' di magia: Java possiede una funzionalità detta *garbage collector*, che tiene sotto controllo tutti gli oggetti creati con l'operatore new, per determinare quelli che non sono più referenziati; quindi rilascia la memoria occupata da questi oggetti affinché possa essere riutilizzata da altri. Ciò implica che non dovete mai preoccuparvi di recuperare memoria. Limitatevi a creare gli oggetti; quando non ne avrete più bisogno spariranno da soli. In questo modo, Java elimina un'intera classe di problemi: le cosiddette “perdite di memoria” (*memory leak*), che si verificano tipicamente quando lo sviluppatore dimentica di liberare la memoria occupata.



Creazione di nuovi tipi di dato: class

Se tutto è un oggetto, che cosa determina il comportamento di una particolare classe di oggetti? In altre parole, che cosa stabilisce il *tipo* di un oggetto? Potreste attendervi l'esistenza di una parola chiave "type", e questo avrebbe assolutamente senso.

Tuttavia, molti linguaggi "storici" orientati agli oggetti hanno previsto la parola chiave **class** per indicare: "tra poco vedrai a che cosa corrisponde un nuovo tipo di oggetto". La parola chiave **class**, comune al punto che non verrà sempre evidenziata in grassetto, è seguita dal nome del nuovo tipo. Per esempio:

```
class ATypeName { /* Qui va inserito il corpo della classe */ }
```

Questa dichiarazione introduce un nuovo tipo; il corpo della classe non permette di fare molto, contenendo soltanto un commento (i simboli di commento verranno trattati più avanti, in questo capitolo), ma è comunque possibile creare un oggetto del tipo appena creato, utilizzando **new**:

```
ATypeName a = new ATypeName();
```

Tuttavia non potete ordinargli di fare alcunché, in altre parole non potete inviargli nessun messaggio interessante, fino a quando non definirete alcuni metodi.

Campi e metodi

Quando definite una classe, e tutto ciò che fate in Java è definire classi (oltre a creare oggetti da queste classi e inviare messaggi a questi oggetti), potete includere due tipi di elementi: i *campi*, chiamati talvolta *membri dati*, e i *metodi*, noti anche come *funzioni membro*.

Un campo è un oggetto di un tipo arbitrario, con cui dialogare tramite il suo riferimento, oppure un tipo primitivo.

Se è un riferimento a un oggetto, dovete inizializzare il riferimento per collegarlo a un oggetto effettivo, ricorrendo all'operatore **new** visto in precedenza.

Ogni oggetto mantiene l'archivio dei propri campi; i campi normali non vengono condivisi tra gli oggetti. Questo è un esempio di classe contenente alcuni campi:

```
class DataOnly {  
    int i;  
    double d;  
    boolean b;  
}
```

Questa classe, pur limitandosi a contenere dati, permette di creare un oggetto analogo al seguente:

```
DataOnly data = new DataOnly();
```

Potete assegnare valori ai campi, ma prima dovete sapere come fare riferimento a un membro di un oggetto. Questa operazione si esegue dichiarando il nome del riferimento dell'oggetto, seguito da un punto (.) e dal nome del membro contenuto nell'oggetto:

```
objectReference.member
```

Per esempio:

```
data.i = 47;  
data.d = 1.1;  
data.b = false;
```

È anche possibile che il vostro oggetto contenga altri oggetti, i quali, a loro volta, potrebbero contenere dati da modificare. In questi casi dovete semplicemente "collegare i punti", per esempio:

```
myPlane.leftTank.capacity = 100;
```

La classe **DataOnly** non fa altro che contenere dati poiché non possiede alcun metodo. Per comprendere come lavorare con i metodi dovete prima assimilare i concetti di *argomenti* e di *valori di ritorno*, descritti tra breve.

Valori predefiniti per i membri primitivi

Quando il membro di una classe è un tipo di dati primitivo, se non lo inizializzate Java gli assegnerà un valore predefinito.

Tipo primitivo	Valore predefinito
boolean	False
char	'\u0000' (null)
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d

Java garantisce questi valori predefiniti soltanto quando la variabile viene utilizzata come *membro di una classe*. Questo comportamento riduce una possibile fonte di errori, assicurando che le variabili membro di tipi primitivi vengano sempre inizializzate, operazione che C++ non esegue. Tuttavia, questo valore iniziale potrebbe essere sbagliato o persino illegale per il programma che state scrivendo: è quindi preferibile che inizializziate esplicitamente le variabili che vi occorrono.

L'assegnazione di un valore predefinito non si applica invece alle variabili locali, vale a dire quelle che non sono campi di una classe. Quindi, se all'interno della definizione del metodo vi fosse

```
int x;
```

la variabile **x** otterebbe un valore arbitrario, come in C / C++, senza essere automaticamente inizializzata a zero: avete la responsabilità di assegnare a **x** un valore appropriato prima di utilizzarlo. In ogni caso, quando dimenticate di inizializzare la variabile Java ha un comportamento migliore rispetto a C++: visualizza un errore di compilazione, facendovi notare che la variabile potrebbe non essere stata inizializzata. Tenete presente che molti compilatori C++ ricorrono a un semplice *warning* per avvertirvi della presenza di variabili non inizializzate, ma in Java questo è considerato un errore.

Metodi, argomenti e valori di ritorno

Molti linguaggi, come C e C+, utilizzano il termine *funzione* per descrivere una certa subroutine: Java ricorre più comunemente al termine *metodo*, inteso come “modo per realizzare qualcosa”. Se lo preferite potete continuare a pensare in termini di funzioni: effettivamente si tratta soltanto di una differenza sintattica, tuttavia in questo manuale si è ritenuto di adottare il termine “metodo”, di uso comune in Java.

I metodi Java determinano i messaggi che un oggetto può ricevere. I componenti fondamentali di un metodo sono il nome, gli argomenti, il tipo di ritorno e il corpo. La sintassi di base è la seguente:

```
ReturnType methodName( /* Elenco degli argomenti */ ) {
    /* Corpo del metodo */
}
```

Il tipo di ritorno definisce il valore restituito dal metodo dopo la chiamata; l'elenco degli argomenti indica i tipi e i nomi delle informazioni che volete passare al metodo. Il nome del metodo e l'elenco dei suoi argomenti si combinano nella cosiddetta *segnatura* o *firma* (*signature*) del metodo, che lo identifica in modo univoco.

In Java i metodi possono essere creati soltanto come parte di una classe. Un metodo può essere chiamato soltanto per un oggetto, e quest'ultimo deve essere in grado di eseguire la chiamata al metodo.⁴

Se cercate di chiamare un metodo che non esiste nell'oggetto otterrete un messaggio di errore in fase di compilazione. Per chiamare un metodo di un oggetto indicate il nome dell'oggetto seguito da un punto (.), a sua volta seguito dal nome del metodo e dall'elenco dei suoi argomenti, come in:

```
objectName.methodName(arg1, arg2, arg3);
```

Per esempio, supponete di avere un metodo **f()** che non accetta argomenti e restituisce un valore di tipo **int**. In questo caso, se avete un oggetto **a** per cui possa essere chiamato **f()**, potrete usare questa sintassi:

```
int x = a.f();
```

4. In realtà, come vedrete tra breve, i metodi **static** possono essere chiamati per la classe e non richiedono un oggetto.



Il tipo del valore di ritorno deve essere compatibile con il tipo della variabile `x`.

Di solito in Java la chiamata di un metodo è indicata con l'espressione "inviare un messaggio a un oggetto". Nell'esempio precedente il messaggio è `f()` e l'oggetto è `a`. La programmazione a oggetti viene spesso riassunta semplicemente come "l'invio di messaggi agli oggetti".

Elenco degli argomenti

L'elenco di argomenti dei metodi specifica quali informazioni passare al metodo. Anche queste informazioni, come tutto in Java, assumono la forma di oggetti. Pertanto, nell'elenco dovete specificare il tipo di oggetto da passare e il nome da utilizzare per ogni argomento. Come accade spesso in Java, quando vi sembra di gestire oggetti in realtà state utilizzando riferimenti: in ogni caso il tipo assegnato al riferimento deve essere corretto.⁵

Se si suppone che l'argomento sia una stringa (oggetto `String`), dovete passare una stringa, altrimenti vi sarà segnalato un errore di compilazione.

Considerate un metodo che richiede una stringa come argomento; di seguito ne è riportata la definizione, che deve essere inserita all'interno di una definizione di classe affinché il metodo stesso venga compilato:

```
int storage(String s) {
    return s.length() * 2;
}
```

Questo metodo restituisce il numero di byte necessari per contenere le informazioni in una determinata stringa (`String`). In questo caso specifico notate che la dimensione di ogni `char` in una `String` è di 16 bit, o due byte, per supportare i caratteri Unicode. L'argomento è di tipo `String` ed è chiamato `s`. Dopo aver passato `s` al metodo potrete gestirlo esattamente come qualsiasi altro oggetto, per esempio inviandogli messaggi. Nell'esempio viene chiamato il metodo `length()`, uno dei metodi disponibili per gli oggetti `String`, che restituisce il numero di caratteri presenti nella stringa.

Avrete notato anche l'uso della parola chiave `return`, che ha un duplice scopo: innanzitutto significa "esci dal metodo, perché ho finito"; inoltre, se il metodo deve restituire un valore, questo viene indicato dopo la parola chiave



`return`. Nell'esempio riportato il valore di ritorno è prodotto dalla valutazione dell'espressione `s.length() * 2`.

Durante la progettazione di un metodo potete scegliere di restituire qualsiasi tipo di dato desideriate; se volete che il metodo non restituisca alcun valore, dovete comunque specificare che esso restituisce `void`. Alcuni esempi sono i seguenti:

```
boolean flag() { return true; }
double naturalLogBase() { return 2.718; }
void nothing() { return; }
void nothing2() {}
```

Quando il valore di ritorno è `void`, la parola chiave `return` viene usata soltanto per uscire dal metodo; non è quindi necessaria quando si raggiunge la fine del metodo. Potete utilizzare `return` in ogni punto del metodo, tuttavia, se avete previsto un tipo di ritorno diverso da `void`, opportuni messaggi di errore del compilatore vi "costringeranno" a restituire il tipo appropriato di valore, indipendentemente dal punto di uscita.

A questo punto un programma potrebbe essere considerato un semplice insieme di oggetti con metodi che prendono come argomenti altri oggetti, e inviano loro messaggi. In sostanza, questo è quasi tutto ciò di cui dovete occuparvi; nel capitolo successivo, tuttavia, imparerete a eseguire in dettaglio le attività di basso livello, prendendo le opportune decisioni all'interno di un metodo. Per questo capitolo, la nozione di invio dei messaggi è sufficiente.

Come costruire un programma Java

Vi sono altri argomenti che dovete comprendere prima di affrontare il vostro primo programma Java.

Visibilità del nome

Un problema comune a tutti i linguaggi di programmazione è il controllo dei nomi degli oggetti.

Se utilizzate un nome in un modulo del programma e un altro sviluppatore sceglie lo stesso nome in un altro modulo, come potete distinguere un nome dall'altro e impedire che "entrino in collisione"? In C, in particolare, questo è un problema grave, poiché un programma spesso è una marea ingestibile di nomi. Le classi C++, su cui si basano quelle di Java, annidano le funzioni

5. Con l'eccezione dei tipi di dato "speciali", già descritti: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` e `double`. In generale, comunque, passerete oggetti, ovvero riferimenti agli oggetti stessi.

all'interno di classi, cosicché esse non possano essere in contrasto con nomi di funzione presenti in altre classi. Tuttavia C++ ammette ancora la presenza di dati e funzioni globali, pertanto i conflitti sono ancora possibili. Per risolvere questo problema il linguaggio C++ ha introdotto l'utilizzo degli spazi di nomi (*namespace*), ricorrendo a parole chiave aggiuntive.

Java è stato in grado di evitare tutto questo, adottando un approccio inedito. Per produrre un nome non ambiguo per una libreria, gli ideatori di Java hanno previsto che utilizziate il vostro nome di dominio Internet, scritto al contrario, dal momento che i nomi di dominio sono sicuramente univoci.

Per esempio, considerato che il nome di dominio dell'autore è **MindView.net**, la sua libreria di utilità *foibles* si chiamerebbe **net.mindview.utility.foibles**. Notate che dopo il nome di dominio invertito i punti rappresentano le sottodirectory.

In Java 1.0 e Java 1.1 le estensioni di dominio **com, edu, org, net** ecc. per convenzione erano in lettere maiuscole, cosicché il nome della libreria sarebbe stato **NET.mindview.utility.foibles**. A un certo punto dello sviluppo di Java 2, tuttavia, si è notato che questa norma causava problemi, pertanto ora l'intero nome del pacchetto è in minuscolo.

Questo meccanismo significa che tutti i vostri file esistono automaticamente nel proprio spazio di nomi, e che ogni classe all'interno di un file deve avere un identificativo univoco; il linguaggio si incarica per voi di evitare conflitti di nomi.

Utilizzo di altri componenti

Ogni volta che volete usare una classe predefinita nel vostro programma il compilatore deve sapere come individuarla. Naturalmente la classe potrebbe già esistere nello stesso file sorgente da cui viene chiamato; in tal caso utilizzerete semplicemente il nome della classe, anche se questa verrà definita soltanto in seguito; Java elimina il problema del cosiddetto "forward referencing".

Ma che cosa accade se una classe è presente in un altro file? Potreste pensare che il compilatore sia abbastanza "intelligente" da riuscire a trovarla, ma qui iniziano i problemi. Immaginate di voler utilizzare una classe con un nome specifico, della quale esistano però più definizioni, presumibilmente diverse l'una dall'altra. Oppure, situazione ben peggiore, immaginate di scrivere un programma e di aggiungere alla vostra libreria una nuova classe che sia in conflitto con il nome di una classe preesistente.

Al fine di risolvere questo tipo di problemi occorre eliminare tutte le ambiguità potenziali. Dovrete quindi segnalare esattamente al compilatore Java quali classi desiderate utilizzare, tramite la parola chiave **import**, che indica al compilatore di introdurre un pacchetto (*package*), vale a dire una libreria di classi. Tenete presente che in altri linguaggi una libreria potrebbe essere composta di funzioni, dati e classi, ma ricordate anche che tutto il codice Java deve essere scritto all'interno di classi.

Il più delle volte vi servirete di componenti provenienti dalle librerie standard di Java, fornite con il compilatore. Per questi componenti non dovete preoccuparvi dei lunghi nomi di dominio invertiti, ma per esempio indicate semplicemente

```
import java.util.ArrayList;
```

per indicare al compilatore che volete usare la classe **ArrayList** di Java. Tuttavia **util** contiene molte altre classi; potreste voler utilizzarne diverse, senza per questo doverle elencare in modo esplicito. Per fare questo potete usare il carattere jolly **"*"**:

```
import java.util.*;
```

Tenete presente che è prassi comune importare un insieme di classi in questo modo piuttosto che singolarmente.

La parola chiave static

Di norma, quando create una classe state anche descrivendone l'aspetto e il comportamento. In effetti, non disponete di un oggetto fino a quando non ne create uno utilizzando l'operatore **new**: è a questo punto che viene allocato in memoria e ne sono disponibili i metodi.

Esistono tuttavia due situazioni in cui tale approccio non è sufficiente. Una di queste si verifica quando volete avere un singolo punto di archiviazione per un determinato campo, a prescindere da quanti oggetti di questa classe sono stati creati, o se non ne è stato creato nessuno. L'altro caso è quando avete bisogno di un metodo che non sia associato a nessun oggetto particolare di questa classe: in altre parole, quando vi occorre un metodo richiamabile anche se non è creato alcun oggetto.

Entrambe queste esigenze possono essere soddisfatte dalla parola chiave **static**. Affermare che un elemento è di tipo **static** significa che quel particolare campo o metodo non è legato ad alcuna istanza particolare di un oggetto



di quella classe. Pertanto, anche senza aver creato nessun oggetto di quella classe potrete chiamare un metodo **static** o accedere a un campo **static**.

Quando avete a che fare con normali campi e metodi non **static** dovete per prima cosa creare un oggetto, da utilizzare poi per accedere al campo o al metodo: i campi e i metodi non **static** devono conoscere lo specifico oggetto con cui operano.⁶

Alcuni linguaggi orientati agli oggetti adottano i termini *variabili di classe* (*class data*) e *metodi di classe* (*class method*), indicando così che tali dati e metodi esistono solo per la classe in generale, non per qualsiasi particolare oggetto di quella stessa classe. Anche nella documentazione dedicata a Java, talvolta, si fa riferimento a questi termini.

Per rendere **static** un campo o un metodo, è sufficiente anteporre questa parola chiave alla definizione. Il codice seguente, per esempio, crea una variabile statica (**static**) e la inizializza:

```
class StaticTest {  
    static int i = 47;  
}
```

Ora, anche creando due oggetti **StaticTest**, sarà disponibile soltanto un'area di archiviazione per la variabile **StaticTest.i**: entrambi gli oggetti condivideranno lo stesso **i**. Prendete in esame l'esempio seguente:

```
StaticTest st1 = new StaticTest();  
StaticTest st2 = new StaticTest();
```

Quindi **st1.i** e **st2.i** avranno lo stesso valore, 47, dal momento che si riferiscono alla stessa area di memoria.

Esistono due tecniche per fare riferimento a una variabile **static**. Come mostrato nell'esempio precedente è possibile richiamarla per mezzo di un oggetto, come in **st2.i**. In alternativa potete farvi riferimento direttamente, tramite il nome della classe, operazione che non potete invece eseguire con un membro non **static**.

```
StaticTest.i++;
```

L'operatore **++** incrementa di un'unità la variabile; quindi, ora **st1.i** e **st2.i** avranno il valore 48.

L'utilizzo del nome della classe è il metodo preferito per fare riferimento a una variabile **static**: infatti evidenzia la natura statica della variabile e in alcuni casi fornisce al compilatore migliori opportunità di ottimizzazione.

Una logica simile si applica ai metodi **static**: potete farvi riferimento sia attraverso un oggetto, come con qualsiasi metodo, sia mediante la sintassi speciale **ClassName.method()**. Un metodo **static** viene definito nel modo seguente:

```
class Incrementable {  
    static void increment() { StaticTest.i++; }  
}
```

Come potete notare, il metodo **increment()** della classe **Incrementable** aumenta il valore della variabile statica **i** servendosi dell'operatore **++**. Questo metodo può essere chiamato nel modo consueto:

```
Incrementable sf = new Incrementable();  
sf.increment();
```

Oppure, dal momento che **increment()** è un metodo **static**, potete chiamarlo direttamente tramite la sua classe:

```
Incrementable.increment();
```

Malgrado la parola chiave **static**, se applicata a una variabile, cambi profondamente il modo in cui i dati vengono creati (uno per ogni classe contro uno per ogni oggetto, tipico delle versioni non **static**), se applicata a un metodo non ha effetti così notevoli.

La principale applicazione di **static** rimane comunque quella di consentirvi la chiamata del metodo senza creare un oggetto. Come vedrete, questo è essenziale nella definizione del metodo **main()**, il punto di inizio dell'esecuzione di un programma.

6. Naturalmente, poiché non richiedono la preventiva creazione di oggetti, i metodi **static** non possono accedere in modo diretto a membri o metodi non **static** semplicemente chiamando questi elementi, senza far riferimento a un determinato oggetto: questo perché i membri e metodi non **static** devono essere collegati a uno specifico oggetto.



Il vostro primo programma Java

Infine ecco il primo programma completo, che inizia visualizzando una stringa, seguita dalla data, ricorrendo alla classe **Date** della libreria standard Java.

```
// HelloDate.java
import java.util.*;

public class HelloDate {
    public static void main(String[] args) {
        System.out.println("Hello, it's: ");
        System.out.println(new Date());
    }
}
```

All'inizio di ogni file sorgente dovete inserire tutte le dichiarazioni **import** necessarie per introdurre le classi supplementari che vi occorreranno. Notate il termine "supplementari": in effetti, esiste una libreria di classi che vengono importate automaticamente in ogni file Java, **java.lang**.

Avviate il browser web e consultate la documentazione di Sun: se non avete ancora scaricato la documentazione JDK da <http://java.sun.com>, fatelo al più presto.⁷

Tenete presente che questa documentazione non è contenuta nel pacchetto JDK e va quindi scaricata a parte. Osservando la lista dei package vedrete tutte le librerie di classi fornite con Java: selezionate **java.lang**. Sarà visualizzato un elenco di tutte le classi che appartengono a quella libreria. Poiché **java.lang** è inclusa implicitamente in ogni file sorgente Java, tutte queste classi si rendono automaticamente disponibili. In **java.lang** non troverete nessuna classe **Date**, quindi dovete importare un'altra libreria per utilizzarla. Se non sapete in quale libreria si trova una particolare classe, o se preferite vedere tutte le classi, nella documentazione Java selezionate il collegamento *Tree*: vi apparirà l'elenco di tutte le classi fornite con Java, e potrete utilizzare la funzione di ricerca del browser per trovare **Date**. La troverete catalogata come **java.util.Date**, a indicare che **Date** si trova nella libreria **util** e che è necessario importare **java.util.*** per utilizzarla.

7. Il compilatore Java e la documentazione Sun vengono aggiornati con una certa frequenza, pertanto è sempre opportuno ottenerli direttamente dal sito di Sun, in modo da disporre della versione più recente.



Se tornate all'inizio, selezionate **java.lang** e poi **System**: vedrete che la classe **System** ha numerosi campi, e se scegliete **out** scoprirete che si tratta di un oggetto **PrintStream static**.

Trattandosi di un oggetto **static** non avete bisogno di creare nulla con **new**. L'oggetto **out** è sempre presente, dovete semplicemente usarlo; ciò che potete fare con l'oggetto **out** è determinato dal suo tipo: **PrintStream**. Per praticità **PrintStream** nella descrizione appare sotto forma di collegamento ipertestuale; facendo clic su di esso verrà visualizzato un elenco di tutti i metodi richiamabili per **PrintStream**. Come vedete sono numerosi, e verranno esaminati nel prosieguo del manuale. Per ora tutto quello che vi occorre è il metodo **println()**, usando il quale è come se ordinaste a Java: "visualizza quello che ti chiedo sul terminale di console, e termina con un carattere di nuova riga". In effetti, in qualsiasi programma Java potete scrivere qualcosa di simile a:

```
System.out.println("A string of things");
```

ogni volta che volete visualizzare informazioni sulla console.

Il nome della classe è identico al nome del file; quando create un programma di tipo *standalone* come questo, infatti, è necessario che una delle classi nel file abbia lo stesso nome del file: in caso contrario il compilatore segnalerà un errore. Tale classe deve contenere un metodo chiamato **main()**, con segnatura e tipo analoghi a:

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

Come vedrete meglio nel Capitolo 6, dedicato al controllo degli accessi, la parola chiave **public** indica che il metodo è disponibile a tutti. L'argomento di **main()** è un array di oggetti **String**; **args** non viene utilizzato in questo programma, tuttavia il compilatore Java ne richiede la presenza perché contiene gli argomenti provenienti dalla riga di comando.

La riga con l'istruzione per visualizzare la data è piuttosto interessante:

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

L'argomento è un oggetto **Date**, creato unicamente per inviare a **println()** il proprio valore, convertito automaticamente in **String**. Non appena questa istruzione termina, l'oggetto **Date** non è più necessario: non dovete preoccuparvi di eliminarlo perché il meccanismo di *garbage collection* può prenderlo in carico in qualsiasi momento.



Esaminando la documentazione JDK ottenuta da <http://java.sun.com> vedrete che **System** dispone di molti altri metodi che permettono di realizzare effetti interessanti: uno dei maggiori punti di forza di Java è appunto la grande quantità di librerie standard. Prendete in esame l'esempio seguente:

```
//: object>ShowProperties.java

public class ShowProperties {
    public static void main(String[] args) {
        System.getProperties().list(System.out);
        System.out.println(System.getProperty("user.name"));
        System.out.println(
            System.getProperty("java.library.path"));
    }
} ///:~
```

La prima riga del metodo **main()** mostra tutte le “proprietà” del sistema sul quale state eseguendo il programma, fornendovi le informazioni di ambiente. Il metodo **list()** trasmette i risultati al suo argomento, **System.out**. Vedrete in seguito che tali risultati possono essere inviati anche altrove, per esempio in un file. Potete anche richiedere proprietà specifiche: in questo esempio, il nome utente e la proprietà **java.library.path**. Scoprirete tra poco il significato degli insoliti commenti all'inizio e alla fine del file.

Compilazione ed esecuzione

Per compilare ed eseguire questo e tutti gli altri programmi presentati nel volume dovete disporre di un ambiente di programmazione Java. Esistono diversi ambienti di sviluppo forniti da altri produttori, ma per questo manuale si è scelto di utilizzare JDK (*Java Developer's Kit*) di Sun, un software gratuito. Se utilizzate un altro sistema di sviluppo, consultate la relativa documentazione per determinare come compilare ed eseguire i programmi.⁸

Collegatevi a Internet e andate sul sito <http://java.sun.com>, dove troverete informazioni e collegamenti che vi guideranno nel processo di download e installazione del JDK per la vostra piattaforma specifica.

8. Il compilatore “jikes” di IBM è un’alternativa comune a JDK, in quanto notevolmente più rapido del *javac* di Sun, benché qualora dobbiate assemblare gruppi di file con *Ant* la differenza di prestazioni non sia così rilevante. Esistono anche progetti open source per la creazione di compilatori, ambienti di runtime e librerie Java.



Dopo aver installato JDK e impostato nella variabile **PATH** le informazioni di percorso, in modo che il vostro sistema possa individuare automaticamente i programmi **javac** e **java**, scaricate e decomprimete il codice sorgente di questo manuale, disponibile all’indirizzo www.mindview.net. Verrà così creata una sottodirectory per ogni capitolo di questo libro; spostatevi nella sottodirectory **object** e digitate il comando:

```
javac HelloDate.java
```

Questo comando non dovrebbe produrre alcun output; qualora otteniate messaggi d’errore è possibile che non abbiate installato correttamente JDK, e dovete quindi investigare a fondo il problema. Di contro, se il prompt di comando non mostra errori potrete digitare:

```
java HelloDate
```

Otterrete come risultato il messaggio di benvenuto e la data corrente. Questo è il procedimento da utilizzare per compilare ed eseguire tutti i programmi presentati in questo libro. Noterete tuttavia che il codice sorgente del manuale ha anche un file chiamato **build.xml** per ogni capitolo: questo file contiene comandi “Ant” per eseguire la compilazione automatica dei file del capitolo. I cosiddetti *buildfile* e Ant (<http://ant.apache.org/>) sono descritti con maggiore dettaglio nel supplemento che troverete all’indirizzo <http://mindview.net/Books/BetterJava>; dopo aver installato Ant potrete digitare semplicemente **ant** al prompt di comando per compilare ed eseguire i programmi di ciascun capitolo. Se non avete ancora installato Ant, dovete eseguire manualmente i comandi **javac** e **java**.

Commenti e documentazione incorporata

Java riconosce due tipi di commenti. Il primo è quello tradizionale in stile C, ereditato anche da C++; questo commento inizia con “*/**” e prosegue, anche su più righe, fino a “**/*”. Tenete presente che molti programmatori fanno precedere da ***** ogni riga di un commento continuato, pertanto avrete spesso occasione di incontrare commenti simili al seguente:

```
/* Questo è un commento
 * che prosegue
 * su diverse righe
 */
```



Ricordate, comunque, che tutto ciò che si trova tra `/*` e `*/` viene ignorato; non noterete alcuna differenza scrivendo:

```
/* Questo è un commento che prosegue  
su più righe */
```

La seconda forma di commento proviene da C++: è il cosiddetto commento di riga, che inizia con `//` e continua fino al termine della riga. Questo tipo di commento è pratico e utilizzato comunemente per la sua semplicità; non richiede che scorriate i tasti alla ricerca del simbolo `*`: vi basta premere il tasto `/` per due volte, e non dovete neppure ricordarvi di chiudere il commento. Troverete quindi spesso commenti simili a:

```
// Questo è un commento di riga
```

Commento di documentazione

Probabilmente il problema principale della documentazione del codice è la sua manutenzione. Se documentazione e codice sono separati, diventa oneroso modificare la documentazione ogni volta che si interviene sul codice. La soluzione più semplice è collegare codice e documentazione, e il modo più facile per farlo consiste nell'includere tutto nello stesso file. Naturalmente una simile tecnica prevede una sintassi di commento speciale per contrassegnare la documentazione, e l'utilizzo di uno strumento per estrarre questi commenti e convertirli in un formato utile: questo è ciò che ha fatto Java.

L'utility per estrarre i commenti è chiamata *Javadoc* e fa parte dell'installazione JDK: ricorrendo in parte alla tecnologia del compilatore Java, questo software ricerca i marcatori (*tag*) dei commenti speciali che vengono inseriti nei programmi. Oltre a estrarre le informazioni contrassegnate da questi tag, Javadoc cattura anche il nome della classe o del metodo che si trova in prossimità del commento. Grazie a Javadoc, con una minima attività siete in grado di realizzare una dignitosa documentazione del codice.

Il risultato prodotto da Javadoc è un file HTML che potete visualizzare nel browser. Javadoc vi permette di creare e gestire un solo file di codice sorgente, generando automaticamente la documentazione utile. Javadoc mette a vostra disposizione uno strumento molto pratico per creare la documentazione, offrendovi la possibilità di integrare anche quella di tutte le librerie Java.

Inoltre, se desiderate eseguire operazioni speciali sulle informazioni trattate da Javadoc potete scrivere voi stessi degli handler, chiamati *doclet*, che vi consentiranno, per esempio, di produrre l'output in un formato diverso. I doclet sono trattati nel supplemento che potete trovare all'indirizzo <http://mindview.net/Books/BetterJava>.

Di seguito viene fornita una panoramica di Javadoc e sono introdotte le sue funzionalità di base: la descrizione completa del software è disponibile nella documentazione JDK.

Dopo aver decompresso la documentazione cercate nella sottodirectory *tool-docs*, o seguite il collegamento Tool Docs presente nella documentazione.

Sintassi

Tutti i comandi Javadoc vengono inseriti soltanto all'interno di commenti `/**`, che si chiudono, come sempre, con `*/`. I metodi principali per l'utilizzo di Javadoc sono due: l'inclusione diretta di codice HTML o l'uso dei cosiddetti "doc tag". I *doc tag standalone* sono comandi che iniziano con `@`, posti all'inizio di una riga di commento: tenete presente che un eventuale `*` iniziale verrà ignorato. I *doc tag inline*, invece, possono apparire ovunque all'interno di un commento Javadoc e anche iniziare con `@`, ma sono racchiusi da parentesi graffe.

Esistono tre "tipi" di commento di documentazione, corrispondenti all'elemento che precede il commento: classe, campo o metodo. In pratica, un commento di classe appare subito prima della definizione di una classe, un commento di campo prima della definizione della variabile e un commento di metodo prima della definizione di un metodo. Osservate questo semplice esempio:

```
//: object/Documentation1.java  
/** Un commento di classe */  
public class Documentation1 {  
    /** Un commento di campo */  
    public int i;  
    /** Un commento di metodo */  
    public void f() {}  
} ///:~
```

Ricordate che in modalità predefinita Javadoc elabora la documentazione solo per i membri **public** e **protected**: i commenti per i membri **private** e



package-access/default (si veda il Capitolo 6, Controllo di accesso) vengono ignorati, pertanto non produrranno alcun output. Se lo desiderate, potete servirvi dell'opzione **-private** per includere nell'elaborazione anche i membri **private**; la scelta predefinita è comunque sensata, in quanto solo i membri **public** e **protected** sono disponibili all'esterno del file, il che è perfettamente conforme alla prospettiva del programmatore client.

Il risultato generato dal codice precedente è un file HTML formattato secondo gli standard che contraddistinguono tutta la documentazione Java: gli utenti si troveranno a loro agio e non avranno difficoltà a consultare le vostre classi.

Vi raccomandiamo di provare il codice dell'esempio precedente, di elaborarlo attraverso Javadoc e di esaminare il file HTML ottenuto per osservarne i risultati.

Inclusione diretta di codice HTML

Javadoc si limita a passare le istruzioni HTML al documento HTML che sarà generato, una caratteristica che vi consente di utilizzare appieno le funzionalità di questo linguaggio di marcatura; in ogni caso il motivo principale per includere inserti HTML è consentire la formattazione del codice, come nell'esempio seguente:

```
//: object/Documentation2.java
/**
* <pre>
* System.out.println(new Date());
* </pre>
*/
public class Documentation2 {}
//:~
```

Potete anche servirvi dell'HTML come fareste in qualsiasi altro documento web, per formattare il testo delle vostre descrizioni:

```
//: object/Documentation3.java
/**
* Potete <em>persino</em> inserire un elenco:
* <ol>
* <li> Voce uno
```



```
* <li> Voce due
* <li> Voce tre
* </ol>
*/
public class Documentation3 {}
//:~
```

Osservate che all'interno del commento di documentazione gli asterischi di inizio riga vengono ignorati da Javadoc, insieme agli eventuali spazi iniziali. Javadoc riformatta tutto il contenuto in modo che sia conforme all'aspetto della documentazione standard. Ricordate di non utilizzare intestazioni come **<h1>** o **<hr>** nell'HTML incorporato, in quanto Javadoc inserisce le proprie intestazioni che interferirebbero con il vostro codice.

Tutti i tipi di commento di documentazione, classe, campo e metodo supportano il codice HTML incorporato.

Alcuni tag di esempio

Di seguito sono descritti alcuni tag Javadoc disponibili per la documentazione del codice. Prima di dedicarvi seriamente all'utilizzo di Javadoc dovreste consultare il manuale di riferimento nella documentazione JDK, al fine di conoscere tutte le possibilità e le modalità di utilizzo di questo strumento.

@see

Questo tag consente di fare riferimento alla documentazione in altre classi. Javadoc genererà il codice HTML con il tag **@see** come collegamento ipertestuale alla documentazione dell'altra classe. Le sintassi ammesse sono le seguenti:

```
@see classname
@see fully-qualified-classname
@see fully-qualified-classname#method-name
```

Ciascuna di queste forme aggiunge un collegamento **See Also** alla documentazione prodotta. Tenete presente che Javadoc non controlla la validità dei collegamenti ipertestuali forniti.



{@link package.class#member label}

Questo tag è molto simile a **@see**, tranne che per il fatto di poter essere utilizzato in linea e di utilizzare il valore **label** come testo del collegamento ipertestuale, in luogo di **See Also**.

{@docRoot}

Genera il percorso relativo per la directory radice della documentazione: è utile per creare collegamenti ipertestuali esplicativi alle pagine nell'alberatura della documentazione.

{@inheritDoc}

Eredita la documentazione dalla classe di base più vicina alla classe corrente, nel commento di documentazione corrente.

@version

La sintassi di questo tag è nel formato

```
@version version-information
```

in cui **version-information** rappresenta qualsiasi informazione importante riteniate di includere. Quando nella riga di comando Javadoc si specifica l'opzione **-version**, le informazioni di versione vengono generate nella documentazione HTML.

@author

La sintassi di questo tag è nel formato

```
@author author-information
```

in cui **author-information** corrisponde solitamente al vostro nome, ma potrebbe anche includere il vostro indirizzo di posta elettronica o qualsiasi altra informazione riteniate appropriata. Se specificate l'opzione **-author** nella riga di comando Javadoc le informazioni sull'autore verranno generate nella documentazione HTML.

È possibile avere numerosi tag **@author**, per presentare un elenco di autori, avendo cura che tali tag siano consecutivi. Tutte le informazioni relative agli autori sono riunite in un solo paragrafo nel codice HTML generato.



@since

Questo tag consente di indicare la versione del codice con la quale avete introdotto una particolare caratteristica o funzionalità. Il valore corrispondente apparirà nella documentazione HTML Java per indicare la versione JDK utilizzata.

@param

Questo tag è utilizzato per la documentazione dei metodi, nella sintassi

```
@param parameter-name description
```

in cui **parameter-name** è l'identificativo nell'elenco di parametri del metodo e **description** è un testo descrittivo, che può anche continuare sulle righe successive. La descrizione è considerata completata quando si incontra un nuovo tag di documentazione. Potete inserire un numero arbitrario di tag, uno per ogni parametro.

@return

Questo tag è utilizzato per la documentazione dei metodi, nella sintassi

```
@return description
```

in cui **description** rappresenta il significato del valore di ritorno, che può continuare su più righe.

@throws

Le eccezioni verranno trattate nel Volume 2, Capitolo 1 dedicato alla gestione degli errori. In sintesi, si tratta di oggetti che possono essere "sollevati" da un metodo, quando esso incontra un problema. Benché durante la chiamata a un metodo possa comparire un solo oggetto di eccezione, un determinato metodo potrebbe produrre un numero qualsiasi di eccezioni di tipi diversi, ciascuna delle quali deve essere descritta.

La sintassi di questo tag è

```
@throws fully-qualified-class-name description
```

in cui **fully-qualified-class-name** rappresenta un nome univoco di una classe di eccezione, definita nel codice, mentre **description**, che può continuare su



diverse righe, contiene informazioni sul motivo per cui un particolare tipo di eccezione può emergere dalla chiamata al metodo.

@deprecated

Questo tag è utilizzato per segnalare caratteristiche che sono state sostituite da nuove funzionalità migliorate: si tratta di un chiaro invito a non servirsi più della funzionalità specifica perché in futuro verrà probabilmente rimossa. Qualora il codice faccia uso di un metodo contrassegnato come `@deprecated` il compilatore genererà un avviso (*warning*). In Java SE5 il tag `@deprecated` di Javadoc è stato sostituito dall'annotazione `@Deprecated`, che potrete esaminare in dettaglio nel Volume 2, Capitolo 8 dedicato alle annotazioni.

Esempio di documentazione

Ecco di nuovo il vostro primo programma Java, ora integrato con i commenti di documentazione:

```
//: object/HelloDate.java
import java.util.*;

/** Il primo programma d'esempio di Thinking in Java.
 * Visualizza una stringa e la data odierna.
 * @author Bruce Eckel
 * @author www.MindView.net
 * @version 4.0
 */
public class HelloDate {
    /** Punto d'ingresso per la classe e l'applicazione.
     * @param args array di argomenti stringa
     * @throws exceptions Nessuna eccezione sollevata
    */
    public static void main(String[] args) {
        System.out.println("Hello, it's:");
        System.out.println(new Date());
    }
} /* Output: (55% match)
Hello, it's:
Wed Oct 05 14:39:36 MDT 2005
*///:~
```

2 • Tutto è un oggetto

La prima riga del file mostra la tecnica dell'autore, che usa `//:` come un segnaposto speciale, contenente il nome archivio del file sorgente. Questa riga contiene il percorso del file, seguito dal nome di file; `object` indica il capitolo che state leggendo. Anche l'ultima riga termina con un commento, nel formato `/// :~`, che indica la fine del listato; questo accorgimento consente di aggiornare automaticamente il codice di questo manuale dopo essere stato controllato con un compilatore ed eseguito.

Il tag `/* Output:` indica l'inizio dell'output generato da questo file: in questa forma è possibile eseguire test automatici per verificarne l'accuratezza.

In questo caso specifico il valore `(55% match)` segnala al sistema di verifica che il risultato sarà alquanto diverso da un'esecuzione all'altra, e che pertanto non ci si deve attendere una corrispondenza superiore al 55% con l'output registrato in questo tag. Quasi tutti gli esempi di questo manuale che producono un output ne contengono anche la versione commentata, in modo che possiate confrontarla con i vostri risultati al fine di verificarne la correttezza.

Stile di codifica

Lo stile definito nel documento *Code Conventions for the Java Programming Language* raccomanda di scrivere in maiuscolo la prima lettera del nome di una classe.⁹

Se il nome della classe è composto di parole multiple, esse vengono unite senza ricorrere a caratteri di sottolineatura (o *underscore*), e la prima lettera di ciascuna parola viene scritta in maiuscolo, come nell'esempio seguente:

```
class AllTheColorsOfTheRainbow { // ...}
```

Questo stile viene indicato con il termine di *UpperCamelCase*, perché l'alternanza di maiuscole e minuscole ricorda le gobbe dei cammelli e perché la prima lettera è maiuscola.

Per quasi tutti gli altri elementi, metodi, campi (variabili membro) e nomi dei riferimenti agli oggetti, lo stile ammesso è identico a quello delle classi, tranne che nella prima lettera dell'identificativo, che deve essere minuscola: questo stile è noto anche come *lowerCamelCase*. Per esempio:

⁹. Il documento è consultabile all'indirizzo <http://java.sun.com/docs/codeconv/index.html>. Per ragioni di spazio, in questo libro e nelle presentazioni dei seminari non è sempre possibile attenersi scrupolosamente alle indicazioni di Sun: in ogni caso noterete che lo stile utilizzato dall'autore è il più possibile conforme agli standard Java.

```
class AllTheColorsOfTheRainbow {
    int anIntegerRepresentingColors;
    void changeTheHueOfTheColor(int newHue) {
        // ...
    }
    // ...
}
```

Considerate che anche l'utente deve scrivere questi nomi così lunghi e descrittivi, pertanto evitate di essere eccessivamente prolissi. Anche la disposizione delle parentesi graffe adottata in questo manuale rispecchia quella che troverete nel codice Java delle librerie Sun.

Riepilogo

L'obiettivo di questo capitolo era quello di presentarvi Java, in modo da insegnarvi a scrivere un semplice programma. Vi è stata presentata una panoramica sul linguaggio e alcuni dei suoi concetti fondamentali. Tuttavia, tutti gli esempi esaminati fino a questo punto sono stati del tipo: "fate questo, poi quello, infine fate quest'altro". I due capitoli che seguono tratteranno gli operatori di base usati in Java e mostreranno come controllare il flusso del programma.

Esercizi

In genere gli esercizi del volume sono distribuiti all'interno dei diversi capitoli; in questo capitolo, tuttavia, avete appreso come scrivere programmi di base, pertanto tutti gli esercizi previsti sono stati rimandati alla fine. Il numero tra parentesi di ogni esercizio segnala il livello di difficoltà, in una scala da 1 a 10.

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

Esercizio 1 (2) Create una classe che contiene un **int** e un **char** non inizializzati, quindi visualizzatene i valori per verificare l'inizializzazione predefinita eseguita da Java.

Esercizio 2 (1) Seguendo l'esempio **HelloDate.java** di questo capitolo, create un programma "Ciao, mondo!" che visualizza semplicemente

questa stringa. In questa classe vi occorrerà soltanto un metodo: il **main()** che viene eseguito all'avvio del programma; ricordate di renderlo **static** e che dovete includere l'elenco degli argomenti, anche se non vengono utilizzati. Compilate il programma con il comando **javac** ed eseguitelo con **java**. Se ritenete di utilizzare un ambiente di sviluppo diverso da JDK, consultate la documentazione per compilare ed eseguire i programmi.

Esercizio 3 (1) Nel capitolo corrente trovate i frammenti di codice relativi a **ATypeName** e trasformateli in un programma compilabile ed eseguibile.

Esercizio 4 (1) Trasformate i frammenti di codice relativi a **DataOnly** in un programma compilabile ed eseguibile.

Esercizio 5 (1) Modificate l'esercizio precedente in modo che i valori dei dati in **DataOnly** vengano assegnati e visualizzati in **main()**.

Esercizio 6 (2) Scrivete un programma che include e chiama il metodo **storage()**, definito come frammento di codice in questo capitolo.

Esercizio 7 (1) Trasformate i frammenti di codice **Incrementable** in un programma funzionante.

Esercizio 8 (3) Scrivete un programma in grado di dimostrare che, a prescindere dal numero di oggetti istanziati di una determinata classe, esiste solo un valore per un campo **static** per quella classe.

Esercizio 9 (2) Scrivete un programma che dimostra il funzionamento della funzionalità *autoboxing* per tutti i tipi primitivi e i rispettivi wrapper.

Esercizio 10 (1) Trasformate l'esempio **AllTheColorsOfTheRainbow** in un programma compilabile ed eseguibile.

Esercizio 11 (2) Trovate il codice della seconda versione di **HelloDate.java**, che è l'esempio dei commenti di documentazione. Eseguite **Javadoc** sul file ed esamineate i risultati con il browser web.

Esercizio 12 (1) Eseguite **Documentation1.java**, **Documentation2.java** e **Documentation3.java** tramite **Javadoc**. Verificate la documentazione risultante con il browser web.

Esercizio 13 (1) Aggiungete un elenco di voci in formato HTML alla documentazione dell'esercizio precedente.

Esercizio 14 (1) Prendete il programma sviluppato nell'Esercizio 2 e aggiungetevi i commenti di documentazione. Estraete questa docu-



mentazione in un file HTML utilizzando **Javadoc** ed esaminatela con il browser web.

Esercizio 15 (1) Nel Capitolo 5, dedicato all'inizializzazione e al cleanup, individuate l'esempio **Overloading.java** e aggiungetevi la documentazione Javadoc. Estraete questa documentazione in un file HTML utilizzando **Javadoc** ed esaminatela con il browser web.

Capitolo 3

Gli operatori



Al livello più basso, la manipolazione dei dati in Java avviene mediante l'utilizzo degli operatori.

Poiché Java ha ereditato da C++, i programmati di questi linguaggi troveranno familiare la maggior parte di tali operatori, che Java ha in qualche modo migliorato e semplificato.

Se conoscete a fondo la sintassi di C e C++ potrete limitarvi a scorrere questo capitolo e quello successivo, soffermandovi soltanto sulle differenze tra Java e questi linguaggi. Di contro, se ritenete che la lettura di questi capitoli sia difficile, non esitate a seguire il seminario multimediale *Thinking in C*, liberamente scaricabile dal sito www.mindview.net: esso contiene letture audio, diapositive, esercizi e soluzioni progettati specificamente per fornirvi rapidamente le nozioni di base necessarie per l'apprendimento di Java.

La dichiarazione più semplice: print

Nel capitolo precedente avete potuto familiarizzare con la prima operazione di visualizzazione in Java:

```
System.out.println("Rather a lot to type");
```



Avrete probabilmente notato che questa riga non è soltanto molto lunga da digitare, ma anche piuttosto noiosa da leggere. La maggior parte dei linguaggi precedenti e successivi a Java ha adottato un approccio più semplice per le istruzioni di utilizzo tanto comune.

Il Capitolo 6, dedicato al controllo di accesso, introduce il concetto di importazione statica (*static import*), integrata in Java SE5, e crea una piccola libreria per semplificare le operazioni di scrittura delle istruzioni di visualizzazione. Per iniziare a utilizzare questa libreria, tuttavia, non avete bisogno di conoscere dettagli particolari, ma potete farlo fin d'ora e riscrivere il programma del capitolo precedente:

```
//: operators/HelloDate.java
import java.util.*;
import static net.mindview.util.Print.*;

public class HelloDate {
    public static void main(String[] args) {
        print("Hello, it's:");
        print(new Date());
    }
} /* Output: (55% match)
Hello, it's:
Wed Oct 05 14:39:05 MDT 2005
*//*:~
```

Il risultato è un codice sicuramente più leggibile: notate l'inserimento della parola chiave **static** nella seconda istruzione **import**.

Per usare questa libreria dovete scaricare il pacchetto contenente il codice del manuale da www.mindview.net o da uno dei suoi siti mirror. Decomprimete l'alberatura delle directory con il codice e aggiungete la directory radice alla variabile di ambiente *CLASSPATH* del vostro computer: tra breve verrete introdotti ai misteri di questa variabile, in ogni caso dovete abituarvi il più presto possibile a "combattere" con essa, poiché questa sarà una delle battaglie più ricorrenti nell'utilizzo di Java.

Nonostante l'utilizzo di **net.mindview.util.Print** semplifichi in parte il codice, non sempre è giustificabile: se il programma contiene poche istruzioni di visualizzazione, per esempio, si può tranquillamente evitare l'importazione e scrivere l'istruzione completa **System.out.println()**.



Esercizio 1 (1) Scrivete un programma che utilizza l'istruzione di visualizzazione nelle forme "breve" e standard.

Utilizzo degli operatori Java

Un operatore elabora uno o più argomenti e produce un nuovo valore. Gli argomenti sono in forma diversa da quella di una comune chiamata di metodo, ma l'effetto è identico. In tutti i linguaggi di programmazione, somma e segno positivo unario (+), sottrazione e segno negativo unario (-), moltiplicazione (·), divisione (/) e assegnazione (=) si comportano allo stesso modo.

Tutti gli operatori producono un valore dai loro operandi; inoltre, alcuni operatori cambiano il valore di un operando, generando quello che viene chiamato *effetto collaterale*. L'utilizzo più comune degli operatori che modificano i propri operandi è appunto la creazione di questo effetto collaterale, ma ricordate che il valore prodotto è anch'esso disponibile all'uso, esattamente come avviene con gli operatori privi di effetti collaterali.

Quasi tutti gli operatori funzionano soltanto con i tipi primitivi; le eccezioni sono costituite da "=", "==", "!=" che funzionano con tutti gli oggetti e possono essere fonte di confusione. Inoltre, la classe **String** supporta "+" e "+=".

Precedenza

La precedenza degli operatori definisce come valutare un'espressione in cui sono presenti più operatori. Java ha una serie di regole specifiche che determinano l'ordine di valutazione degli operatori: la regola più semplice da ricordare è che moltiplicazione e divisione vengono elaborate prima della somma e della sottrazione. Tuttavia, spesso i programmati dimenticano le altre regole di precedenza; in caso di dubbio è opportuno che utilizziate le parentesi per rendere esplicito l'ordine di valutazione. Analizzate, per esempio, le dichiarazioni (1) e (2):

```
//: operators/Precedence.java
```

```
public class Precedence {
    public static void main(String[] args) {
        int x = 1, y = 2, z = 3;
        int a = x + y - 2/2 + z;           // (1)
        int b = x + (y - 2)/(2 + z);     // (2)
```



```

System.out.println("a = " + a + " b = " + b);
}
} /* Output:
a = 5 b = 1
*///:~

```

A una semplice occhiata queste dichiarazioni sembrano identiche, ma osservando il prodotto potete vedere che hanno significati molto diversi, che dipendono dall'uso delle parentesi.

Si noti l'uso dell'operatore “+” all'interno dell'istruzione **System.out.println()**. In questo contesto “+” significa “concatenazione” e, se necessario, “conversione in stringa”. Quando il compilatore incontra una **String** seguita da un segno “+”, seguito a sua volta da una variabile non **String**, tenta di convertire quest'ultima in una stringa. Come potete vedere dall'output, la conversione avviene con successo da **int** in **String** per le variabili **a** e **b**.

Assegnazione

L'assegnazione viene eseguita mediante l'operatore **=**. Questa operazione significa “prendere il valore sul lato destro, spesso indicato come *rvalue*, e copiarlo nel lato sinistro, chiamato anche *lvalue*”. Un *rvalue* può essere qualsiasi costante, variabile o espressione che produca un valore, mentre un *lvalue* deve essere una variabile distinta e nominata: in altri termini, deve esistere uno spazio fisico in cui conservare il valore. Per esempio, potete assegnare un valore costante a una variabile come nell'esempio seguente:

```
a = 4;
```

Non potete tuttavia assegnare alcunché a una costante: in pratica, una costante non può essere un *lvalue*, e non potete dire, per esempio, **4 = a;**

L'assegnazione dei tipi primitivi è piuttosto semplice. Poiché il tipo primitivo conserva il valore effettivo e non un riferimento a un oggetto, quando assegnate valori ai tipi primitivi, in pratica copiate il contenuto da una posizione a un'altra. Per esempio, scrivendo **a = b**, il contenuto di **b** viene copiato in **a**. Se proseguite modificando **a**, la variabile **b** non verrà influenzata da questa modifica. Come programmatore, questo è ciò che potete aspettarvi in molte situazioni.

Quando eseguite assegnazioni con gli oggetti, tuttavia, le cose cambiano. Ogni volta che manipolate un oggetto, in realtà state lavorando con il suo riferimento, pertanto quando assegnate “da un oggetto a un altro”, in effetti state copiando un



riferimento da un'area a un'altra. Questo significa che per gli oggetti, scrivendo **c = d**, ordinate a **c** e **d** di puntare verso l'oggetto che in origine indicava soltanto **d**. Ecco un esempio che dimostra questo comportamento:

```

//: operators/Assignment.java
// L'assegnazione con gli oggetti e' meno semplice
// di quanto sembri.
import static net.mindview.util.Print.*;

class Tank {
    int level;
}

public class Assignment {
    public static void main(String[] args) {
        Tank t1 = new Tank();
        Tank t2 = new Tank();
        t1.level = 9;
        t2.level = 47;
        print("1: t1.level: " + t1.level +
              ", t2.level: " + t2.level);
        t1 = t2;
        print("2: t1.level: " + t1.level +
              ", t2.level: " + t2.level);
        t1.level = 27;
        print("3: t1.level: " + t1.level +
              ", t2.level: " + t2.level);
    }
} /* Output:
1: t1.level: 9, t2.level: 47
2: t1.level: 47, t2.level: 47
3: t1.level: 27, t2.level: 27
*///:~

```

La classe **Tank** è molto chiara: all'interno di **main()** vengono create due istanze, **t1** e **t2**. Ai campi **level** di ogni **Tank** viene assegnato un valore diverso; poi **t2** è assegnato a **t1** e **t1** viene modificato. In molti linguaggi di pro-

grammazione ci si aspetterebbe che **t1** e **t2** siano sempre indipendenti; invece, dal momento che avete assegnato un riferimento, cambiando l'oggetto **t1** si modifica anche **t2**. Questo avviene perché **t1** e **t2** contengono lo stesso riferimento, che indica lo stesso oggetto. Il riferimento originale che si trovava in **t1**, che puntava all'oggetto contenente il valore 9, è stato sovrascritto durante l'assegnazione e quindi perso; il relativo oggetto è stato preso in carico dal garbage collector.

Questo fenomeno viene anche chiamato *aliasing*, e fondamentalmente è il modo in cui Java opera con gli oggetti. Se non volete che si verifichi l'aliasing potete estendere l'assegnazione in questo modo:

```
t1.level = t2.level;
```

Così facendo i due oggetti rimarranno distinti, invece di scartarne uno e legare le variabili **t1** e **t2** allo stesso oggetto. Presto vi renderete conto che manipolare i campi all'interno di oggetti è un'operazione che può potenzialmente generare confusione, e che va contro i principi di progettazione orientati agli oggetti. Si tratta di un argomento tutt'altro che banale: dovete ricordare che l'assegnazione per oggetti può dar luogo a sorprese.

Esercizio 2 (1) Create una classe contenente un **float** e utilizzatela per dimostrare l'aliasing.

Aliasing in fase di chiamata di metodo

L'aliasing si verifica anche passando un oggetto a un metodo:

```
//: operators/PassObject.java
// Il passaggio degli oggetti ai metodi potrebbe
// essere diverso da ciò' cui eravate abituati.
import static net.mindview.util.Print.*;

class Letter {
    char c;
}

public class PassObject {
    static void f(Letter y) {
        y.c = 'z';
    }
}
```

```
public static void main(String[] args) {
    Letter x = new Letter();
    x.c = 'a';
    print("1: x.c: " + x.c);
    f(x);
    print("2: x.c: " + x.c);
}
} /* Output:
1: x.c: a
2: x.c: z
*///:~
```

In molti linguaggi di programmazione il metodo **f()** produrrebbe una copia del suo **Letter** **y** nell'ambito del metodo. Ancora una volta tuttavia viene passato un riferimento, cosicché la riga

```
y.c = 'z';
```

in realtà cambia l'oggetto fuori da **f()**.

Il problema dell'aliasing e la sua soluzione rappresentano un argomento complesso. In ogni caso, ora siete consapevoli di questo problema e dovete essere preparati alle trappole che potrebbero presentarsi.

Esercizio 3 (1) Create una classe contenente un **float** e usatela per dimostrare l'aliasing durante le chiamate dei metodi.

Operatori matematici

Gli operatori matematici di base sono gli stessi disponibili nella maggior parte dei linguaggi di programmazione: somma (+), sottrazione (-), divisione (/), moltiplicazione (·) e modulo (%), che restituisce il resto di una divisione di valori interi. Tenete presente che la divisione tra interi tronca il risultato, anziché arrotondarlo.

Java permette di utilizzare la notazione abbreviata tipica di C/C++, che esegue simultaneamente l'operazione e l'assegnazione. L'espressione tipica è caratterizzata da un operatore seguito da un segno di uguale, ed è coerente con tutti gli operatori del linguaggio (ogni volta che ha senso eseguire questa operazione).

Per esempio, per aggiungere 4 alla variabile **x** e assegnare il risultato a **x**, ossia incrementare di 4 la variabile **X**, si può utilizzare l'espressione **x += 4**. Questo esempio dimostra l'utilizzo degli operatori matematici:

```
//: operators/MathOps.java
// Dimostra l'uso degli operatori matematici.
import java.util.*;
import static net.mindview.util.Print.*;

public class MathOps {
    public static void main(String[] args) {
        // Crea un generatore di numeri casuali,
        // passando un seme:
        Random rand = new Random(47);
        int i, j, k;
        // Sceglie un valore tra 1 e 100:
        j = rand.nextInt(100) + 1;
        print("j : " + j);
        k = rand.nextInt(100) + 1;
        print("k : " + k);
        i = j + k;
        print("j + k : " + i);
        i = j - k;
        print("j - k : " + i);
        i = k / j;
        print("k / j : " + i);
        i = k * j;
        print("k * j : " + i);
        i = k % j;
        print("k % j : " + i);
        j %= k;
        print("j %= k : " + j);
        // Verifica valori in virgola mobile:
        float u, v, w; // Si applica anche ai double
        v = rand.nextFloat();
        print("v : " + v);
```

```
w = rand.nextFloat();
print("w : " + w);
u = v + w;
print("v + w : " + u);
u = v - w;
print("v - w : " + u);
u = v * w;
print("v * w : " + u);
u = v / w;
print("v / w : " + u);
// Il codice seguente funziona anche
// con i tipi char, byte, short, int,
// long e double:
u += v;
print("u += v : " + u);
u -= v;
print("u -= v : " + u);
u *= v;
print("u *= v : " + u);
u /= v;
print("u /= v : " + u);
}
} /* Output:
j : 59
k : 56
j + k : 115
j - k : 3
k / j : 0
k * j : 3304
k % j : 56
j %= k : 3
v : 0.5309454
w : 0.0534122
v + w : 0.5843576
v - w : 0.47753322
v * w : 0.028358962
```

```
v / w : 9.940527
u += v : 10.471473
u -= v : 9.940527
u *= v : 5.2778773
u /= v : 9.940527
*///:~
```

Per generare numeri, il programma crea innanzitutto un oggetto **Random**. Se create un oggetto **Random** senza argomenti, Java utilizza l'orario corrente come seme per il generatore di numeri casuali, producendo quindi un risultato diverso a ogni esecuzione del programma. Tuttavia, negli esempi di questo manuale è importante che il prodotto mostrato alla fine del codice sorgente sia il più coerente possibile, in modo da essere verificabile con l'ausilio di strumenti esterni. Se durante la creazione dell'oggetto **Random** si fornisce un seme, ovvero un valore d'inizializzazione per il generatore di numeri casuali, esso produrrà gli stessi numeri casuali a ogni esecuzione del programma, rendendo così l'output facilmente accertabile. Al fine di ottenere risultati più casuali, non esitate a rimuovere il seme inserito negli esempi nel libro.

Tramite l'oggetto **Random** il programma genera molti numeri casuali di vario tipo, chiamando semplicemente i metodi **nextInt()** e **nextFloat()**; è possibile usare anche i metodi **nextLong()** e **nextDouble()**. L'argomento fornito al metodo **nextInt()** imposta il limite superiore del numero casuale generato; il limite inferiore è zero, che non è opportuno utilizzare poiché potrebbe dar luogo a un errore *divide-by-zero*, cosicché il risultato viene compensato da uno.

Esercizio 4 (2) Scrivete un programma che calcola la velocità utilizzando una distanza e un tempo costanti.

Operatori unari di segno positivo e negativo

Il segno negativo unario (**-**, o *meno unario*) e il segno positivo unario (**+**, o *più unario*) sono gli stessi operatori utilizzati per le operazioni binarie di sottrazione e somma. Il compilatore identifica l'utilizzo che intendete fare di questi operatori in base a sintassi dell'espressione. Per esempio, la dichiarazione

```
x = -a;
```

ha un significato evidente. Invece, sebbene il compilatore non abbia problemi a interpretare

```
x = a * -b;
```

il lettore potrebbe confondersi, pertanto spesso risulta più chiaro scrivere:

```
x = a * (-b);
```

Il segno meno unario inverte il segno del dato. Il segno più unario esiste soltanto per ragioni di simmetria con il meno, e il suo unico effetto è trasformare in **int** tipi di operandi più piccoli.

Incremento e decremento automatici

Come il linguaggio C, Java dispone di molte scorciatoie o abbreviazioni che possono rendere il codice più facile da digitare, e più o meno difficile da leggere.

Le due abbreviazioni più interessanti riguardano gli operatori di incremento e di decremento, noti anche come operatori di auto-incremento e di auto-decremento. L'operatore di decremento è **--** e significa "diminuzione di un'unità"; l'operatore di incremento è **++**, e corrisponde ad "aumento di un'unità". Per esempio, se **a** è un **int** l'espressione **++a** equivarrà ad **(a = a + 1)**. Gli operatori di decremento e incremento non si limitano a modificare la variabile, ma producono come risultato il valore della variabile stessa.

Esistono due versioni per ogni tipo di operatore, spesso denominate *suffissee postfisse*: il *pre-incremento* indica che l'operatore **++** compare prima della variabile, mentre nel *post-incremento* l'operatore **++** segue la variabile; in modo analogo, nel *pre-decremento* e nel *post-decremento* gli operatori compaiono rispettivamente prima e dopo la variabile. Con il *pre-incremento* e il *pre-decremento* (come in **++a** e **--a**) viene prima eseguita l'operazione e poi prodotto il risultato; invece, nel *post-incremento* e nel *post-decremento* (come in **a++** e **a--**) viene prima generato il valore, poi eseguita l'operazione. Osservate questo esempio:

```
//: operators/AutoInc.java
// Dimostra l'uso degli operatori ++ e --.
import static net.mindview.util.Print.*;

public class AutoInc {
    public static void main(String[] args) {
        int i = 1;
        print("i : " + i);
        print("++i : " + ++i); // Pre-incremento
```



```

print("i++ : " + i++); // Post-incremento
print("i : " + i);
print("--i : " + --i); // Pre-decremento
print("i-- : " + i--); // Post-decremento
print("i : " + i);
}
} /* Output:
i : 1
++i : 2
i++ : 2
i : 3
--i : 2
i-- : 2
i : 1
*///:~

```

Entrambe le forme incrementano o decrementano l'operando cui sono assegnati: se l'operatore è utilizzato nella modalità con prefisso (**++n**), la variabile viene incrementata o decrementata prima di utilizzarne il valore, mentre con la forma con suffisso (**n++**), l'incremento o il decremento della variabile avvengono dopo che ne è stato utilizzato il valore. Questi sono gli unici operatori, se si eccettuano quelli che implicano assegnazione, in grado di produrre un effetto collaterale: cambiano l'operando invece di limitarsi a utilizzarne il valore.

A titolo di curiosità, questo spiega uno dei significati del nome C++, che significa “un passo oltre il C”. In un discorso introduttivo a Java, Bill Joy, uno dei creatori di questo linguaggio, ha affermato che “Java=C++-” (C più più meno meno): questo suggerisce che Java sia una variante C++ cui sono state rimosse le parti più complesse e inutili, al fine di ottenere un linguaggio più semplice. Proseguendo nella lettura di questo manuale vedrete che molte parti sono effettivamente più semplici, mentre in altre Java non è meno ostico di C++.

Operatori relazionali

Gli operatori relazionali producono un risultato booleano valutando la relazione tra i valori degli operandi. Un'espressione relazionale produce **true** (vero) se la relazione è vera e **false** (falso) se è falsa. Gli operatori relazionali sono *minore di (<)*, *maggiore di (>)*, *minore o uguale a (<=)*, *maggiore o uguale a (>=)*, *equi-*



valente a (o uguale a, ==) e *non equivalente (o diverso da, !=)*. L'equivalenza e la non equivalenza sono applicabili a tutti i tipi primitivi, ma gli altri confronti non funzioneranno con il tipo **boolean**: poiché i valori booleani possono essere soltanto vero o falso, “maggiore di” e “minore di” non hanno rilevanza.

Test sull'equivalenza degli oggetti

Gli operatori relazionali **==** e **!=** operano anche con tutti gli oggetti, tuttavia il loro significato confonde spesso i programmati Java alle prime armi. Ecco un esempio:

```

//: operators/Equivalence.java

public class Equivalence {
    public static void main(String[] args) {
        Integer n1 = new Integer(47);
        Integer n2 = new Integer(47);
        System.out.println(n1 == n2);
        System.out.println(n1 != n2);
    }
} /* Output:
false
true
*///:~

```

L'istruzione **System.out.println (n1 == n2)** visualizza il risultato del confronto booleano posto all'interno delle parentesi. A prima vista l'output dovrebbe essere prima **true** e poi **false**, poiché i due oggetti **Integer** sono identici. Tuttavia, mentre il *contenuto* degli oggetti è lo stesso, i *riferimenti* sono diversi: gli operatori **==** e **!=** confrontano i riferimenti agli oggetti, pertanto in realtà il programma produce prima **false** e poi **true**. Naturalmente, questo non manca di sorprendere gli utenti.

Per confrontare il contenuto effettivo di due oggetti dovete utilizzare il metodo speciale **equals()**: esso è disponibile per tutti gli oggetti, ma non per i tipi primitivi, che non danno questo problema con **==** e **!=**. Ecco un esempio del suo impiego:

```

//: operators/EqualsMethod.java

public class EqualsMethod {

```

```

public static void main(String[] args) {
    Integer n1 = new Integer(47);
    Integer n2 = new Integer(47);
    System.out.println(n1.equals(n2));
}
/* Output:
true
*/

```

Ora il risultato è quello che vi aspettereste, tuttavia le cose non sono mai così semplici come sembrano. Se create una classe come

```

//: operators/EqualsMethod2.java
// equals() in modalita' predefinita non confronta i
contenuti.

```

```

class Value {
    int i;
}

public class EqualsMethod2 {
    public static void main(String[] args) {
        Value v1 = new Value();
        Value v2 = new Value();
        v1.i = v2.i = 100;
        System.out.println(v1.equals(v2));
    }
}
/* Output:
false
*/

```

le cose si complicano nuovamente: il risultato è **false**. Questo avviene poiché il comportamento predefinito di **equals()** è di confrontare i riferimenti; quindi, a meno di non sovrascrivere il metodo **equals()** nella vostra nuova classe, non otterrete il comportamento atteso.

Purtroppo non vedrete come sovrascrivere i metodi fino al Capitolo 7 dedicato al riutilizzo delle classi, e apprenderete a definire opportunamente **equals()** sol-

tanto nel Capitolo 5 del Volume 2: in ogni modo, il fatto di essere consapevoli del comportamento di **equals()** nel frattempo potrebbe risparmiarvi qualche problema.

Ricordate che la maggior parte delle librerie di classi di Java implementa **equals()** in modo da confrontare il contenuto di oggetti, non i loro riferimenti.

Esercizio 5 (2) Create una classe chiamata **Dog** contenente due stringhe: "nome" e "verso". Nel metodo **main()** create due oggetti **Dog**: uno di nome "Spot", il cui verso è "Grrrr!", l'altro chiamato "Scruffy", che dice "Bau!". Poi visualizzate i nomi dei due cani e i rispettivi versi.

Esercizio 6 (3) Proseguite l'Esercizio 5, creando un nuovo riferimento **Dog** e assegnandolo all'oggetto **Spot**. Poi eseguite un confronto utilizzando **==** ed **equals()** per tutti i riferimenti.

Operatori logici

I tre operatori logici **AND** (**&&**) **OR** (**||**) e **NOT** (**!**) producono un valore **boolean**, **true** o **false** in base alla relazione logica tra gli argomenti. L'esempio seguente utilizza gli operatori relazionali e logici:

```

//: operators/Bool.java
// Operatori logici e relazionali.
import java.util.*;
import static net.mindview.util.Print.*;

public class Bool {
    public static void main(String[] args) {
        Random rand = new Random(47);
        int i = rand.nextInt(100);
        int j = rand.nextInt(100);
        print("i = " + i);
        print("j = " + j);
        print("i > j is " + (i > j));
        print("i < j is " + (i < j));
        print("i >= j is " + (i >= j));
        print("i <= j is " + (i <= j));
        print("i == j is " + (i == j));
        print("i != j is " + (i != j));
    }
}

```



```
// In Java non e' possibile trattare un int come boolean:  
//! print("i && j is " + (i && j));  
//! print("i || j is " + (i || j));  
//! print("!i is " + !i);  
print("(i < 10) && (j < 10) is "  
    + ((i < 10) && (j < 10)) );  
print("(i < 10) || (j < 10) is "  
    + ((i < 10) || (j < 10)) );  
}  
} /* Output:  
i = 58  
j = 55  
i > j is true  
i < j is false  
i >= j is true  
i <= j is false  
i == j is false  
i != j is true  
(i < 10) && (j < 10) is false  
(i < 10) || (j < 10) is false  
*/:~
```

Potete utilizzare **AND**, **OR** o **NOT** soltanto con valori **boolean**. In un'espressione logica, Java non permette di utilizzare un valore non booleano come se fosse booleano, come è invece possibile fare in C e C++. Nel codice precedente potete vedere i tentativi infruttuosi di eseguire tale operazione, commentati con “**!!**”; questa sintassi di commento permette la rimozione automatica dei commenti al fine di agevolare i test. Le espressioni successive, invece, producono valori booleani ricorrendo a confronti relazionali, utilizzando poi le operazioni logiche su questi risultati.

Tenete presente che un valore **boolean** viene convertito automaticamente in una forma testuale appropriata se è utilizzato dove è richiesta una **String**.

Nel programma precedente potete sostituire la definizione di **int** con qualsiasi altro tipo di dato primitivo, tranne **boolean**. Tenete presente, in ogni caso, che il confronto di numeri in virgola mobile richiede il massimo rigore: come sapete, un numero che sia anche di una minuscola frazione diverso da un altro numero è comunque “non uguale”; un numero che sia di un’infinitesima frazione sopra lo zero è ovviamente diverso da zero, quindi “non zero”.

Esercizio 7 (3) Scrivete un programma che simula il lancio di una moneta.

Cortocircuito

Lavorando con gli operatori logici non è raro imbattersi in un comportamento chiamato “cortocircuito”. Questo termine indica che l'espressione verrà valutata soltanto fino a quando sia determinata la verità o falsità dell'intera espressione: di conseguenza le ultime parti di un'espressione logica potrebbero non essere valutate. Ecco un esempio che dimostra tale comportamento:

```
//: operators/ShortCircuit.java  
// Dimostra il comportamento del cortocircuito  
// con gli operatori logici.  
import static net.mindview.util.Print.*;  
  
public class ShortCircuit {  
    static boolean test1(int val) {  
        print("test1(" + val + ")");  
        print("result: " + (val < 1));  
        return val < 1;  
    }  
    static boolean test2(int val) {  
        print("test2(" + val + ")");  
        print("result: " + (val < 2));  
        return val < 2;  
    }  
    static boolean test3(int val) {  
        print("test3(" + val + ")");  
        print("result: " + (val < 3));  
        return val < 3;  
    }  
    public static void main(String[] args) {  
        boolean b = test1(0) && test2(2) && test3(2);  
        print("expression is " + b);  
    }  
} /* Output:
```

```
test1(0)
result: true
test2(2)
result: false
expression is false
*///:~
```

Ogni test esegue un confronto tra gli argomenti e restituisce **true** o **false**, poi visualizza le informazioni per mostrarvi ciò che è stato chiamato. I test vengono utilizzati nell'espressione seguente:

```
test1(0) && test2(2) && test3(2)
```

A prima vista potreste ritenere che tutte le tre verifiche vengano eseguite, tuttavia l'output dimostra che non è così. Il primo test risulta **true**, quindi la valutazione dell'espressione continua, tuttavia il secondo test genera il risultato **false**: dal momento che di conseguenza l'intera espressione deve essere **false**, perché continuare a valutare il resto dell'espressione? Sarebbe uno spreco di risorse... La presenza del cortocircuito, infatti, permette di ottenere un potenziale incremento di prestazioni, evitando ove possibile di valutare tutte le parti di un'espressione logica.

Valori letterali

Di norma, quando inserite un valore letterale in un programma, il compilatore sa esattamente quale tipo deve essergli assegnato. In alcuni casi, tuttavia, il tipo è ambiguo: quando questo accade potete indirizzare il compilatore fornendogli informazioni supplementari, sotto forma di caratteri associati al valore letterale. Il codice seguente illustra l'utilizzo di questi caratteri:

```
//: operators/Literals.java
import static net.mindview.util.Print.*;
public class Literals {
    public static void main(String[] args) {
        int i1 = 0x2f; // Esadecimale (minuscolo)
        print("i1: " + Integer.toBinaryString(i1));
        int i2 = 0X2F; // Esadecimale (minuscolo)
        print("i2: " + Integer.toBinaryString(i2));
```

```
int i3 = 0177; // Ottale (con zero iniziale)
print("i3: " + Integer.toBinaryString(i3));
char c = 0xffff; // massimo valore char esadecimale
print("c: " + Integer.toBinaryString(c));
byte b = 0x7f; // massimo valore byte esadecimale
print("b: " + Integer.toBinaryString(b));
short s = 0x7fff; // massimo valore short esadecimale
print("s: " + Integer.toBinaryString(s));
long n1 = 200L; // suffisso long
long n2 = 200l; // suffisso long (ma puo' generare
                 // confusione)
long n3 = 200;
float f1 = 1;
float f2 = 1F; // suffisso float
float f3 = 1f; // suffisso float sfixed
double d1 = 1d; // suffisso double
double d2 = 1D; // suffisso double
// (Hex e Octal funzionano anche con i valori long)
}
} /* Output:
i1: 101111
i2: 101111
i3: 1111111
c: 1111111111111111
b: 1111111
s: 1111111111111111
*///:~
```

Un carattere finale dopo un valore letterale ne determina il tipo. Una **L** maiuscola o minuscola indicano il tipo **long**; tuttavia, una lettera **I** minuscola rischierebbe di confondere il programmatore, che potrebbe scambiarla con un numero uno. La lettera **F**, maiuscola o minuscola, indica un **float**. La lettera **D**, maiuscola o minuscola, indica un **double**.

Il formato esadecimale (a base 16), che opera con tutti i tipi di dato interi, è indicato dal prefisso **0x** o **0X** seguito dalle cifre **0-9** o dai caratteri **a-f** maiuscoli o minuscoli. Se cercate di inizializzare una variabile con un valore più grande di quanto possa contenere, a prescindere dalla forma numerica del valore, il compilatore vi segnalerà un messaggio di errore. Nel codice precedente potete notare i possibili

valori esadecimale massimi per i tipi **char**, **byte** e **short**. Se superate questi massimi, il compilatore trasformerà automaticamente il valore in un **int**, segnalandovi la necessità di un *cast* di riduzione per l'assegnazione (argomento che vedrete nel prosieguo del capitolo): in questo modo saprete di aver oltrepassato il limite.

Il formato ottale (a base 8) è caratterizzato da uno zero iniziale e dalle cifre da 0 a 7.

C, C++ e Java non offrono una rappresentazione letterale per i numeri binari. Tuttavia, quando si lavora con le notazioni esadecimale e ottale, può essere utile mostrare i risultati in forma binaria.

Questa operazione è facilmente eseguibile mediante il metodo statico **toBinaryString()**, tipico delle classi **Integer** e **Long**. Tenete presente che passando tipi più piccoli a **Integer.toBinaryString()**, il tipo viene convertito automaticamente in **int**.

Esercizio 8 (2) Mostrate il funzionamento delle notazioni esadecimale e ottale con valori **long**, e utilizzate **Long.toBinaryString()** per visualizzare i risultati.

Notazione esponenziale

Gli esponenti sono rappresentati in una notazione che suscita spavento in molti lettori:

```
//: operators/Exponents.java
// "e" significa "10 elevato alla ... potenza"

public class Exponents {
    public static void main(String[] args) {
        // Le 'e' minuscole e maiuscole sono indifferenti:
        float expFloat = 1.39e-43f;
        expFloat = 1.39E-43f;
        System.out.println(expFloat);
        double expDouble = 47e47d; // 'd' e' opzionale
        double expDouble2 = 47e47; // e' automaticamente double
        System.out.println(expDouble);
    }
} /* Output:
1.39E-43
4.7E48
*///:~
```

In matematica e ingegneria, la “e” indica la base dei logaritmi naturali, corrispondente a circa 2,718. Java, peraltro, offre anche un valore **double** più preciso, **Math.E**.

La “e” viene utilizzata nelle espressioni di elevamento a potenza, per esempio $1,39 \cdot e^{43}$, che corrisponde a $1,39 \cdot 2,718^{43}$.

Purtroppo, quando fu progettato il linguaggio di programmazione FORTRAN venne deciso che “e” avrebbe significato “dieci elevato alla ... potenza” (*ten to the power*). Questa è stata indubbiamente una scelta curiosa, soprattutto tenuto conto che FORTRAN è stato concepito per applicazioni scientifiche e matematiche: dai progettisti ci si sarebbe aspettata una maggiore attenzione nei confronti di una simile ambiguità.¹

In ogni caso, questa consuetudine è stata mantenuta in C e C++, e ora continua in Java. Pertanto, se siete soliti pensare a e come alla base dei logaritmi naturali, dovete fare uno sforzo per adattarvi a un'espressione Java quale **1,39 e-43f**: essa significa infatti $1,39 \cdot 10^{-43}$.

Tenete presente che il carattere finale non è necessario quando il compilatore riesce a identificare il tipo appropriato. In

```
long n3 = 200;
```

non vi sono ambiguità di sorta, quindi la lettera **L** dopo il 200 sarebbe superflua. Invece, con

```
float f4 = 1e-43f; // 10 elevato alla ... potenza
```

1. John Kirkham scrive: “Mi sono avvicinato all'informatica nel 1962, usando FORTRAN II su un computer IBM 1620. A quell'epoca, fino a tutti gli anni Sessanta e Settanta, FORTRAN era un linguaggio scritto interamente in maiuscolo: la ragione di questa particolarità è forse il fatto che molti dei primi dispositivi telescriventi implementavano il codice Baudot a 5 bit, che era privo di minuscole. Anche la “E” nelle notazioni esponenziali veniva indicata sempre in maiuscolo, e non poteva essere confusa con la base dei logaritmi naturali, “e”, sempre in minuscolo. La “E” significava semplicemente *esponenziale*, ovvero la base del sistema numerico utilizzato, di norma 10. A quell'epoca i programmati usavano spesso anche la notazione ottale: sebbene non mi sia mai accaduto, sono certo che se avessi visto un numero ottale in notazione esponenziale lo avrei considerato a base 8. Soltanto negli anni Settanta ho visto per la prima volta un esponenziale scritto con la “e” minuscola, e ricordo che mi aveva lasciato perplesso. Il problema, quindi, si è verificato quando FORTRAN è stato progressivamente contaminato dalle minuscole, non agli inizi. Esistevano già funzioni utilizzabili per lavorare con la base dei logaritmi naturali, ma tutte erano in maiuscolo.”

il compilatore normalmente considera i numeri esponenziali come **double**, cosicché senza la F segnalerebbe un errore per informarvi di utilizzare il cast, e convertire così **double** in **float**.

Esercizio 9 (1) Visualizzate i valori più grandi e più piccoli per i tipi **float** e **double**, in notazione esponenziale.

Gli operatori bit a bit (bitwise)

Gli operatori *bit a bit* o *bitwise* consentono di manipolare singoli bit nei tipi interi primitivi, ed eseguono operazioni booleane sui bit corrispondenti nei due argomenti per produrre il risultato.

L'utilizzo di tali operatori deriva dall'orientamento di basso livello del linguaggio C, in cui spesso è necessario manipolare direttamente l'hardware e impostare i bit nei registri hardware. In origine Java era stato concepito per essere inserito in particolari decoder TV, di conseguenza tale "orientamento di basso livello" era pienamente giustificato. Comunque sia, probabilmente non avrete molte occasioni di utilizzare gli operatori bit a bit.

L'operatore bitwise AND (**&**) restituisce 1 nel bit di output se entrambi i bit di input valgono uno; altrimenti, produce 0. L'operatore bitwise OR (**|**) restituisce 1 nel bit di output se entrambi i bit di input valgono uno, e produce 0 solo se entrambi i bit valgono zero. L'operatore bitwise XOR, od OR ESCLUSIVO (**^**), restituisce 1 se uno qualsiasi dei bit di input vale uno, ma non entrambi. L'operatore bitwise NOT (**~**, detto anche *operatore di complemento a uno*) è di tipo unario in quanto accetta un solo argomento, a differenza di tutti gli altri operatori bit a bit che sono, invece, binari. L'operatore bit a bit NOT restituisce l'opposto del bit di input: 1 se il bit inserito vale zero, e 0 se il bit inserito vale uno.

Gli operatori bit a bit e quelli logici utilizzano gli stessi caratteri, pertanto vi sarà utile ricorrere a un meccanismo mnemonico che vi aiuti a ricordarne il significato, per esempio: poiché i bit sono "piccoli", gli operatori bit a bit sono indicati da un solo carattere.

Gli operatori bit a bit possono essere abbinati al segno = per combinare l'operazione e l'assegnazione: **&=**, **|=** e **^=** sono tutte sintassi legittime.

Il tipo **boolean** è trattato come un valore a un solo bit, quindi si comporta in modo diverso: infatti potete eseguire operazioni bitwise AND, OR e XOR, ma non un NOT bitwise, presumibilmente per evitare confusione con il NOT logico standard. Per i tipi **boolean**, gli operatori bit a bit hanno lo stesso effetto degli operatori logici, tranne che per il meccanismo di cortocircuito. Inoltre le operazioni bit a bit su valori **boolean** includono un operatore logico XOR che non fi-

gura nell'elenco degli operatori "logici". Ricordate che non è possibile utilizzare valori booleani nelle espressioni di shift, descritte di seguito.

Esercizio 10 (3) Scrivete un programma con due valori costanti con valori 0 e 1 alternati: nella cifra meno significativa un valore dovrà contenere 0, l'altro 1. Un suggerimento: per risolvere l'esercizio troverete più pratico servirvi di costanti esadecimali. Combinate questi due valori in tutti i modi possibili utilizzando gli operatori bit a bit, e visualizzate i risultati utilizzando **Integer.toBinaryString()**.

Gli operatori di tipo shift

Gli operatori di spostamento o di scorrimento (*shift operator*) consentono di manipolare anche i bit, e sono utilizzabili esclusivamente con tipi primitivi interi.

L'operatore di shift sinistro (**<<**) sposta a sinistra i bit dell'operando sinistro, per un numero di posizioni pari al numero di bit specificato dall'operando a destra dell'operatore, inserendo valori 0 nei bit di ordine più basso. L'operatore di shift destro con segno (**>>**) sposta a destra i bit dell'operando sinistro, per un numero di posizioni pari al numero di bit specificato dall'operando a destra dell'operatore. L'operatore **>>** utilizza la cosiddetta "estensione di segno" (*sign-extension*): se il valore è positivo, nei bit di ordine elevato viene inserito 0; se il valore è negativo viene inserito 1. Java ha aggiunto anche un operatore che non esiste né in C né in C++: l'operatore di shift destro senza segno (**>>>**), che non utilizza estensioni (*zero-extension*) e, indipendentemente dal segno, inserisce valori 0 nei bit di ordine elevato.

Se eseguite operazioni di spostamento su variabili **char**, **byte** o **short**, esse verranno promosse a **int** prima che l'operazione avvenga, e il risultato sarà un **int**; verranno utilizzati soltanto i cinque bit meno significativi dell'operando destro: questo impedirà di spostare un numero di bit superiore a quelli presenti in un **int**. Se state lavorando con un **long**, otterrete un risultato **long**; verranno usati soltanto i sei bit meno significativi dell'operando destro, in modo che non possiate spostare più del numero di bit presenti in un **long**.

Gli operatori di spostamento possono essere combinati con il segno uguale, al fine di ottenere **<<=**, **>>=** o **>>>=**. Il valore *lvalue* viene sostituito dal valore *lvalue* spostato di *rvalue*. Tuttavia, esiste un problema con lo shift destro senza segno combinato con l'assegnazione: utilizzandolo con **byte** o **short** non otterrete i risultati corretti, poiché dopo essere state promosse a **int** e spostate verso destra, queste variabili vengono troncate al momento di essere riassegnate alle



loro variabili d'origine; in questo caso, otterrete -1. Questo comportamento è dimostrato dall'esempio seguente:



Nell'ultima operazione di shift il valore risultante non viene riassegnato a **b**, ma visualizzato direttamente, così da ottenere il comportamento corretto.

L'esempio che segue, invece, dimostra l'utilizzo di tutti gli operatori che operano sui bit:

```
//: operators/BitManipulation.java
// Utilizzo degli operatori bit a bit.
import java.util.*;
import static net.mindview.util.Print.*;

public class BitManipulation {
    public static void main(String[] args) {
        Random rand = new Random(47);
        int i = rand.nextInt();
        int j = rand.nextInt();
        printBinaryInt("-1", -1);
        printBinaryInt("+1", +1);
        int maxpos = 2147483647;
        printBinaryInt("maxpos", maxpos);
        int maxneg = -2147483648;
        printBinaryInt("maxneg", maxneg);
        printBinaryInt("i", i);
        printBinaryInt("~i", ~i);
        printBinaryInt("-i", -i);
        printBinaryInt("j", j);
        printBinaryInt("i & j", i & j);
        printBinaryInt("i | j", i | j);
        printBinaryInt("i ^ j", i ^ j);
        printBinaryInt("i << 5", i << 5);
        printBinaryInt("i >> 5", i >> 5);
        printBinaryInt("(~i) >> 5", (~i) >> 5);
        printBinaryInt("i >>> 5", i >>> 5);
    }
}
```



```

printBinaryInt("(~i) >>> 5", (~i) >>> 5);

long l = rand.nextLong();
long m = rand.nextLong();
printBinaryLong("-1L", -1L);
printBinaryLong("+1L", +1L);
long ll = 9223372036854775807L;
printBinaryLong("maxpos", ll);
long lln = -9223372036854775808L;
printBinaryLong("maxneg", lln);
printBinaryLong("1", 1);
printBinaryLong("~1", ~1);
printBinaryLong("-1", -1);
printBinaryLong("m", m);
printBinaryLong("l & m", l & m);
printBinaryLong("l | m", l | m);
printBinaryLong("l ^ m", l ^ m);
printBinaryLong("l << 5", l << 5);
printBinaryLong("l >> 5", l >> 5);
printBinaryLong("(~l) >> 5", (~l) >> 5);
printBinaryLong("l >>> 5", l >>> 5);
}

static void printBinaryInt(String s, int i) {
    print(s + ", int: " + i + ", binary:\n" +
        Integer.toBinaryString(i));
}

static void printBinaryLong(String s, long l) {
    print(s + ", long: " + l + ", binary:\n" +
        Long.toBinaryString(l));
}

/* Output:
-1, int: -1, binary:
11111111111111111111111111111111
+1, int: 1, binary:
1
*/

```

```

maxpos, int: 2147483647, binary:
11111111111111111111111111111111
maxneg, int: -2147483648, binary:
10000000000000000000000000000000
i, int: -1172028779, binary:
10111010001001000100001010010101
~i, int: 1172028778, binary:
1000101110110111011110101101010
-i, int: 1172028779, binary:
100010111011011101111010110111
j, int: 1717241110, binary:
1100110010110110000010100010110
i & j, int: 570425364, binary:
100010000000000000000000010100
i | j, int: -25213033, binary:
1111111001111110100011110010111
i ^ j, int: -595638397, binary:
1101110001111110100011110000011
i << 5, int: 1149784736, binary:
1000100100010000101001010100000
i >> 5, int: -36625900, binary:
11111101110100010010001000010100
(~i) >> 5, int: 36625899, binary:
10001011101101110111101011
i >>> 5, int: 97591828, binary:
101110100010010001000010100
(~i) >>> 5, int: 36625899, binary:
10001011101101110111101011
...
*///:~

```

Gli ultimi due metodi, **printBinaryInt()** e **printBinaryLong()**, accettano rispettivamente un **int** e un **long** e li visualizzano in formato binario insieme a una stringa descrittiva. Oltre a dimostrare l'effetto di tutti gli operatori bit a bit per **int** e **long**, questo esempio illustra anche i valori minimo, massimo, +1 e -1 per **int** e **long**, per darvi un'idea della loro rappresentazione in forma

binaria. Notate che il bit alto indica il segno: 0 rappresenta il segno positivo, 1 quello negativo. Nel codice è mostrato l'output per la parte **int**.

La rappresentazione binaria dei numeri è nota anche come “complemento a due con segno” (*signed twos complement*).

Esercizio 11 (3) Iniziate con un numero nel quale il bit più significativo abbia il valore 1. Un suggerimento: utilizzate una costante esadecimale. Servendovi dell'operatore di shift destro con segno eseguite lo spostamento in tutte le sue posizioni binarie e visualizzate ogni volta il risultato con **Integer.toBinaryString()**.

Esercizio 12 (3) Iniziate con un numero contenente solo valori binari 1. Eseguiete lo spostamento a sinistra, poi, mediante l'operatore di shift sinistro senza segno, scorrete verso destra in tutte sue posizioni binarie, visualizzando ogni volta il risultato con **Integer.toBinaryString()**.

Esercizio 13 (1) Scrivete un metodo che mostri i valori **char** in forma binaria e dimostratene il funzionamento servendovi di diversi caratteri.

Operatore ternario: if-else

L'operatore ternario, o condizionale, è atipico poiché si serve di tre operandi.

Si tratta effettivamente di un operatore poiché produce un valore, a differenza del costrutto **if-else** che vedrete nel prossimo paragrafo di questo capitolo. L'espressione assume la forma seguente:

`boolean-exp ? value0 : value1`

Se **boolean-exp** vale **true**, viene valutato **value0** e il suo risultato diventa il valore prodotto dall'espressione. Invece, se **boolean-exp** è **false**, viene valutato **value1** e il suo risultato diventa il valore prodotto dall'espressione.

Naturalmente potreste utilizzare anche una comune istruzione **if-else**, che vedrete in seguito, tuttavia considerate che l'operatore ternario è molto più compatto. Il linguaggio C, in cui ha avuto origine tale operatore, si vanta di essere un linguaggio sintetico, e l'operatore ternario potrebbe essere stato progettato per motivi di efficienza. In ogni caso dovete essere piuttosto prudenti nell'utilizzarlo su base quotidiana, dal momento che rende il codice più difficile da leggere.

L'operatore condizionale è diverso dal costrutto **if-else** poiché produce un valore. L'esempio seguente evidenzia il confronto tra i due meccanismi:

```
//: operators/TernaryIfElse.java
import static net.mindview.util.Print.*;

public class TernaryIfElse {
    static int ternary(int i) {
        return i < 10 ? i * 100 : i * 10;
    }
    static int standardIfElse(int i) {
        if(i < 10)
            return i * 100;
        else
            return i * 10;
    }
    public static void main(String[] args) {
        print(ternary(9));
        print(ternary(10));
        print(standardIfElse(9));
        print(standardIfElse(10));
    }
} /* Output:
900
100
900
100
*///:~
```

È evidente che il codice nel metodo **ternary()** è più conciso di quello scritto senza l'operatore ternario, in **standardIfElse()**. Tuttavia, senza obbligarvi a scrivere molto di più, **standardIfElse()** è di più facile comprensione. Abbiate cura di valutare con attenzione le ragioni che vi portano a preferire l'operatore ternario: in genere è consigliabile utilizzarlo quando sia necessario impostare una variabile a uno di due valori.

Operatori di String + e +=

Java consente un utilizzo speciale degli operatori: come avete visto, + e += possono essere utilizzati per concatenare stringhe. All'apparenza questo è un utilizzo naturale di questi operatori, sebbene non sia coerente con il loro impiego tradizionale.

Questa funzionalità era sembrata una buona idea in C++, di conseguenza in questo linguaggio si è scelto di implementare il cosiddetto *sovraffunzione degli operatori* (*operator overloading*), un meccanismo che consente al programmatore di aggiungere significati alla maggior parte degli operatori. Purtroppo l'overloading, combinato con alcune limitazioni di C++, si è rivelato troppo complesso perché i programmatori possano servirsene nella progettazione delle loro classi. Benché rispetto al C++ l'implementazione di questa funzionalità presenti minori difficoltà, come peraltro è stato dimostrato nel linguaggio C#, in cui l'overloading è un'operazione molto pratica, il sovraffunzione degli operatori è stato giudicato ancora troppo difficile da implementare: ne risulta che gli sviluppatori Java non possono realizzare i propri operatori, a differenza dei programmatori C++ e C#.

L'utilizzo degli operatori **String** presenta alcune caratteristiche interessanti. Se un'espressione inizia con una stringa tutti gli operandi che seguono devono essere stringhe; ricordate che il compilatore trasforma automaticamente una sequenza di caratteri tra virgolette in **String**:

```
//: operators/StringOperators.java
import static net.mindview.util.Print.*;

public class StringOperators {
    public static void main(String[] args) {
        int x = 0, y = 1, z = 2;
        String s = "x, y, z ";
        print(s + x + y + z);
        print(x + " " + s); // Converte x in String
        s += "(summed) = "; // Operatore di concatenamento
        print(s + (x + y + z));
        print("'" + x); // Abbreviazione di Integer.toString()
    }
} /* Output:
   x, y, z 012
   */

```

```
0 x, y, z
x, y, z (summed) = 3
0
*///:~
```

Notate che il prodotto dalla prima istruzione di visualizzazione è “012”, non “3”, che sarebbe stato il risultato della semplice somma di interi; questo avviene perché il compilatore Java converte **x**, **y** e **z** nelle loro rappresentazioni **String** e le concatena, invece di eseguire la somma. La seconda istruzione di visualizzazione converte in modo esplicito la variabile principale in **String**, in modo che l'ordine di conversione non sia casuale. Osservate anche l'utilizzo dell'operatore += che concatena una stringa a **s**, e l'impiego delle parentesi per controllare l'ordine di valutazione dell'espressione, allo scopo di eseguire la somma dei valori **int** prima della loro visualizzazione.

Infine, considerate l'ultimo esempio in **main()**. Talvolta troverete una stringa vuota seguita da un simbolo + e da un tipo primitivo: questo meccanismo esegue la conversione implicita senza chiamare il metodo esplicito, più ingombrante (in questo caso **Integer.toString()**).

Problemi ricorrenti nell'utilizzo degli operatori

Un inconveniente tipico della programmazione può verificarsi omettendo le parentesi negli operatori, qualora si abbia il minimo dubbio sulla loro precedenza, e quindi sul modo in cui verrà valutata l'espressione. Questo è vero anche in Java. Osservate questo tipo di errore, molto comune in C e C++:

```
while(x = y) {
    // ....
}
```

Il programmatore aveva evidentemente intenzione di valutare un'equivalenza (che richiede invece l'utilizzo di ==), non di eseguire un'assegnazione (=). In C e C++ il risultato di tale assegnazione, infatti, è sempre **true** se **y** è diverso da zero; probabilmente eseguendo il codice si otterebbe “soltanto” un ciclo infinito.

In Java, invece, il risultato di tale espressione non è di tipo **boolean**, tuttavia il compilatore si aspetta un risultato booleano e ignorerà il fatto che si tratta di **int**; di conseguenza visualizzerà un errore e rileverà il problema ancora prima che tentiate di eseguire il programma. Java non dà problemi, poiché

una sintassi come `while(x = y)` genererebbe un errore di compilazione: l'unica situazione in cui non si otterrebbe un errore è quando `x` e `y` sono booleani, nel qual caso `x = y` è un'espressione sintatticamente ammessa dal compilatore, ma pur sempre un errore di logica applicativa.

Un problema simile si verifica in C e in C++ utilizzando gli operatori bit a bit AND e OR in sostituzione delle loro versioni logiche. Gli AND e OR bit a bit sono indicati con un solo carattere (`&` e `|`) mentre gli AND e OR logici sono rappresentati da due caratteri (`&&` e `||`). Come con i simboli `=` e `==`, può facilmente accadere di digitare un solo carattere anziché due. In Java è il compilatore a rilevare l'errore, impedendovi di utilizzare il tipo sbagliato.

Operatori di cast

Come già accennato, il termine *cast* è utilizzato nel senso di *to cast*, “colare, replicare in uno stampo”. Se necessario, Java sostituisce automaticamente un tipo di dato in un altro: per esempio, quando assegnate un valore intero a una variabile in virgola mobile il compilatore convertirà l'`int` in un `float`. Mediante l'operazione di *casting* si rende esplicito questo tipo di conversione, o lo si forza qualora non avvenga automaticamente.

Per eseguire il casting specificate tra parentesi il tipo di dato desiderato, a sinistra del valore interessato. In questo esempio potete vedere il casting in azione:

```
//: operators/Casting.java

public class Casting {
    public static void main(String[] args) {
        int i = 200;
        long lNg = (long)i;
        lNg = i; // "Estensione" del tipo: il casting non e'
                  // indispensabile
        long lNg2 = (long)200;
        lNg2 = 200;
        // Un casting di "riduzione" del tipo
        i = (int)lNg2; // Il cast e' necessario
    }
} ///:~
```

Come potete notare, il casting può avvenire sia su un valore numerico sia su una variabile. Tenete presente che nulla vieta di inserire cast ridondanti; per esempio, sebbene il compilatore promuova in modo automatico un valore `int` a uno `long` quando necessario, un cast ridondante potrebbe essere utile per evidenziare o rendere più chiaro il codice. In altre situazioni un cast potrebbe rivelarsi essenziale per la riuscita della compilazione.

In C e C++ il casting può essere causa di emicrania. In Java il casting è un'operazione sicura, con l'eccezione della “conversione con riduzione” (la cosiddetta *narrowing conversion*), che si verifica passando da un tipo di dati più grande a uno più piccolo, con possibile perdita di informazioni. In questi casi il compilatore forza l'utilizzo del casting, dicendo in pratica: “questa operazione può essere rischiosa: se davvero vuoi eseguirla devi rendere esplicito il casting”. Invece, nel caso della “conversione con ampliamento” (*widening conversion*) il cast esplicito non è indispensabile, poiché il nuovo tipo contiene più informazioni del vecchio, pertanto non può esserci perdita di informazioni.

Java consente di eseguire il cast da qualsiasi tipo primitivo a qualsiasi altro tipo primitivo, fatta eccezione per il tipo `boolean`, che non ammette nessun tipo di casting. Analogamente, i tipi di classe non ammettono il casting: per eseguire una conversione di classe dovete ricorrere a metodi speciali. Nel prosieguo del manuale scoprirete che gli oggetti possono essere convertiti nell'ambito di una *famiglia* di tipi: di una **Quercia** si può eseguire il cast in un **Albero**, e viceversa, ma non è possibile farlo in un tipo diverso, per esempio in una **Roccia**.

Troncamento e arrotondamento

Quando eseguite conversioni con riduzione dovete prestare attenzione agli effetti di troncamento e di arrotondamento. Per esempio, se eseguite un cast da un valore in virgola mobile a un valore intero, come si comporterà Java? Supponete di voler eseguire il cast del valore 29,7 in un `int`: il valore risultante sarà 30 o 29? La risposta a questa domanda è nell'esempio seguente:

```
//: operators/CastingNumbers.java
// Che cosa succede convertendo un float
// o un double in un valore intero?
import static net.mindview.util.Print.*;

public class CastingNumbers {
    public static void main(String[] args) {
        double above = 0.7, below = 0.4;
        float fabove = 0.7f, fbelow = 0.4f;
```

```

    print("(int)above: " + (int)above);
    print("(int)below: " + (int)below);
    print("(int)fabove: " + (int)fabove);
    print("(int)fbelow: " + (int)fbelow);
}
/* Output:
(int)above: 0
(int)below: 0
(int)fabove: 0
(int)fbelow: 0
*///:~

```

È evidente che l'operazione di casting da un **float** o da un **double** a un valore intero tronca sempre il valore. Se preferite che il risultato sia arrotondato, utilizzate il metodo **round()** in **java.lang.Math**:

```

//: operators/RoundingNumbers.java
// Arrotondamento di float e double.
import static net.mindview.util.Print.*;

public class RoundingNumbers {
    public static void main(String[] args) {
        double above = 0.7, below = 0.4;
        float fabove = 0.7f, fbelow = 0.4f;
        print("Math.round(above): " + Math.round(above));
        print("Math.round(below): " + Math.round(below));
        print("Math.round(fabove): " + Math.round(fabove));
        print("Math.round(fbelow): " + Math.round(fbelow));
    }
} /* Output:
Math.round(above): 1
Math.round(below): 0
Math.round(fabove): 1
Math.round(fbelow): 0
*///:~

```

Poiché il metodo **round()** è parte di **java.lang** il suo utilizzo non richiede un'importazione supplementare.

Promozione

Eseguendo operazioni matematiche o bit a bit su tipi di dato primitivi più piccoli di un **int** (**char**, **byte** e **short**), vi renderete conto che i valori vengono “promossi” (trasformati) in **int** prima dell'esecuzione delle operazioni e che il valore risultante è di tipo **int**. Pertanto, per assegnare di nuovo il valore al tipo più piccolo dovete utilizzare un cast: in questi casi non dimenticate di tenere conto della possibile perdita di informazioni, visto che state assegnando un valore a un tipo di dato più piccolo. Di norma, il tipo di dato più grande presente in un'espressione determina il tipo del risultato: moltiplicando un **float** e un **double** il risultato sarà **double**, mentre se sommate un **int** e un **long** il risultato sarà **long**.

Java non ha “sizeof”

C e C++ mettono a disposizione l'operatore **sizeof()**, che restituisce il numero di byte assegnato per i dati. Il motivo principale della presenza di **sizeof()** in questi linguaggi è la portabilità. Tipi di dato differenti potrebbero avere dimensioni diverse sui vari hardware: pertanto, quando eseguite operazioni in cui le dimensioni sono un fattore critico, dovete conoscere quelle effettive dei singoli tipi. Per esempio, un computer potrebbe registrare i numeri interi a 32 bit, un altro come valori a 16 bit: sul primo computer, quindi, i programmi potrebbero registrare valori interi più grandi. Come potete immaginare, la portabilità è causa di fastidiose emicranie per i programmatori C e C++.

Pur tenendo conto della portabilità Java non ha bisogno di un operatore **sizeof()**, poiché tutti i tipi di dato hanno dimensioni identiche su tutti i sistemi. Non è necessario considerare la portabilità a questo livello poiché essa è implicita nel linguaggio.

Riepilogo degli operatori

L'esempio che segue mostra quali tipi di dato primitivi è possibile utilizzare con i diversi operatori. In sostanza, si tratta dello stesso esempio ripetuto più volte con i vari tipi di dato primitivi. Il file compilerà senza problemi poiché le righe che generano errori sono state commentate con **!!**.

```

//: operators/AllOps.java
// Verifica tutti gli operatori sui tipi di dato primitivi
// per mostrare quali operazioni sono ammesse dal compilatore.

public class AllOps {

```



```
// Per accettare i risultati di un test booleano:  
void f(boolean b) {}  
  
void boolTest(boolean x, boolean y) {  
    // Operatori aritmetici:  
    //! x = x * y;  
    //! x = x / y;  
    //! x = x % y;  
    //! x = x + y;  
    //! x = x - y;  
    //! x++;  
    //! x--;  
    //! x = +y;  
    //! x = -y;  
  
    // Operatori relazionali e logici:  
    //! f(x > y);  
    //! f(x >= y);  
    //! f(x < y);  
    //! f(x <= y);  
    f(x == y);  
    f(x != y);  
    f(!y);  
    x = x && y;  
    x = x || y;  
  
    // Operatori bit a bit:  
    //! x = ~y;  
    x = x & y;  
    x = x | y;  
    x = x ^ y;  
    //! x = x << 1;  
    //! x = x >> 1;  
    //! x = x >>> 1;  
  
    // Assegnazioni composte:  
    //! x += y;  
    //! x -= y;  
    //! x *= y;  
    //! x /= y;
```

```
//! x %= y;  
//! x <<= 1;  
//! x >= 1;  
//! x >>= 1;  
x &= y;  
x ^= y;  
x |= y;  
  
// Casting:  
//! char c = (char)x;  
//! byte b = (byte)x;  
//! short s = (short)x;  
//! int i = (int)x;  
//! long l = (long)x;  
//! float f = (float)x;  
//! double d = (double)x;  
}  
  
void charTest(char x, char y) {  
    // Operatori aritmetici:  
    x = (char)(x * y);  
    x = (char)(x / y);  
    x = (char)(x % y);  
    x = (char)(x + y);  
    x = (char)(x - y);  
    x++;  
    x--;  
    x = (char)+y;  
    x = (char)-y;  
  
    // Operatori relazionali e logici:  
    f(x > y);  
    f(x >= y);  
    f(x < y);  
    f(x <= y);  
    f(x == y);  
    f(x != y);  
    //! f(!x);  
    //! f(x && y);
```



```
//! f(x || y);
// Operatori bit a bit:
x= (char)~y;
x = (char)(x & y);
x = (char)(x | y);
x = (char)(x ^ y);
x = (char)(x << 1);
x = (char)(x >> 1);
x = (char)(x >>> 1);
// Assegnazioni composte:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <= 1;
x >= 1;
x >>= 1;
x &= y;
x ^= y;
x |= y;
// Casting:
//! boolean bl = (boolean)x;
byte b = (byte)x;
short s = (short)x;
int i = (int)x;
long l = (long)x;
float f = (float)x;
double d = (double)x;
}
void byteTest(byte x, byte y) {
// Operatori aritmetici:
x = (byte)(x* y);
x = (byte)(x / y);
x = (byte)(x % y);
x = (byte)(x + y);
```

```
x = (byte)(x - y);
x++;
x--;
x = (byte)+ y;
x = (byte)- y;
// Operatori relazionali e logici:
f(x > y);
f(x >= y);
f(x < y);
f(x <= y);
f(x == y);
f(x != y);
//! f(!x);
//! f(x && y);
//! f(x || y);
// Operatori bit a bit:
x = (byte)~y;
x = (byte)(x & y);
x = (byte)(x | y);
x = (byte)(x ^ y);
x = (byte)(x << 1);
x = (byte)(x >> 1);
x = (byte)(x >>> 1);
// Assegnazioni composte:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <= 1;
x >= 1;
x >>= 1;
x &= y;
x ^= y;
x |= y;
// Casting:
```



```
//! boolean b1 = (boolean)x;
char c = (char)x;
short s = (short)x;
int i = (int)x;
long l = (long)x;
float f = (float)x;
double d = (double)x;
}

void shortTest(short x, short y) {
    // Operatori aritmetici:
    x = (short)(x * y);
    x = (short)(x / y);
    x = (short)(x % y);
    x = (short)(x + y);
    x = (short)(x - y);
    x++;
    x--;
    x = (short)+y;
    x = (short)-y;
    // Operatori relazionali e logici:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Operatori bit a bit:
    x = (short)~y;
    x = (short)(x & y);
    x = (short)(x | y);
    x = (short)(x ^ y);
    x = (short)(x << 1);
    x = (short)(x >> 1);
```



```
x = (short)(x >>> 1);
// Assegnazioni composte:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <<= 1;
x >>= 1;
x >>>= 1;
x &= y;
x ^= y;
x |= y;
// Casting:
//! boolean b1 = (boolean)x;
char c = (char)x;
byte b = (byte)x;
int i = (int)x;
long l = (long)x;
float f = (float)x;
double d = (double)x;
}

void intTest(int x, int y) {
    // Operatori aritmetici:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Operatori relazionali e logici:
    f(x > y);
    f(x >= y);
```



```
f(x < y);
f(x <= y);
f(x == y);
f(x != y);
//! f(!x);
//! f(x && y);
//! f(x || y);
// Operatori bit a bit:
x = ~y;
x = x & y;
x = x | y;
x = x ^ y;
x = x << 1;
x = x >> 1;
x = x >>> 1;
// Assegnazioni composte:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <= 1;
x >= 1;
x >>= 1;
x &= y;
x ^= y;
x |= y;
// Casting:
//! boolean b1 = (boolean)x;
char c = (char)x;
byte b = (byte)x;
short s = (short)x;
long l = (long)x;
float f = (float)x;
double d = (double)x;
}
```



```
void longTest(long x, long y) {
    // Operatori aritmetici:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Operatori relazionali e logici:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Operatori bit a bit:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    x = x >>> 1;
    // Assegnazioni composte:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <= 1;
```



```
x >>= 1;                                //! x = ~y;
x >>>= 1;                                //! x = x & y;
x &= y;                                    //! x = x | y;
x ^= y;                                    //! x = x ^ y;
x |= y;                                    //! x = x << 1;
// Casting:                                //! x = x >> 1;
//! boolean bl = (boolean)x;                // Assegnazioni composte:
char c = (char)x;                          x += y;
byte b = (byte)x;                          x -= y;
short s = (short)x;                        x *= y;
int i = (int)x;                           x /= y;
float f = (float)x;                        x %= y;
double d = (double)x;                      //! x <<= 1;
}                                         //! x >>= 1;
void floatTest(float x, float y) {          //! x >>>= 1;
    // Operatori aritmetici:                //! x &= y;
    x = x * y;                            //! x ^= y;
    x = x / y;                            //! x |= y;
    x = x % y;                            // Casting:
    x = x + y;                            //! boolean bl = (boolean)x;
    x = x - y;                            char c = (char)x;
    x++;                                 byte b = (byte)x;
    x--;                                 short s = (short)x;
    x = +y;                               int i = (int)x;
    x = -y;                               long l = (long)x;
    // Operatori relazionali e logici:     double d = (double)x;
    f(x > y);                            }
    f(x >= y);                           void doubleTest(double x, double y) {
    f(x < y);                            // Operatori aritmetici:
    f(x <= y);                           x = x * y;
    f(x == y);                           x = x / y;
    f(x != y);                          x = x % y;
    //! f(!x);                           x = x + y;
    //! f(x && y);                      x = x - y;
    //! f(x || y);                      x++;
    // Operatori bit a bit:
```



```

x--;
x = +y;
x = -y;
// Operatori relazionali e logici:
f(x > y);
f(x >= y);
f(x < y);
f(x <= y);
f(x == y);
f(x != y);
//! f(!x);
//! f(x && y);
//! f(x || y);
// Operatori bit a bit:
//! x = ~y;
//! x = x & y;
//! x = x | y;
//! x = x ^ y;
//! x = x << 1;
//! x = x >> 1;
//! x = x >>> 1;
// Assegnazioni composte:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
//! x <= 1;
//! x >= 1;
//! x >>= 1;
//! x &= y;
//! x ^= y;
//! x |= y;
// Casting:
//! boolean b1 = (boolean)x;
char c = (char)x;

```

```

byte b = (byte)x;
short s = (short)x;
int i = (int)x;
long l = (long)x;
float f = (float)x;
}
} //://:~

```

Notate che il tipo **boolean** è abbastanza limitato: consente l'assegnazione dei valori **true** e **false** e le valutazioni su verità e falsità, ma non la somma dei valori **boolean** né altre operazioni.

In **char**, **byte** e **short** potete osservare l'effetto della promozione con gli operatori aritmetici: ogni operazione su uno qualunque di questi tipi produce un risultato **int**, sul quale deve essere eseguito il casting esplicito al tipo originale, cioè una conversione con riduzione che potrebbe causare la perdita di informazioni, per riassegnare il valore al tipo originale. Con i valori **int**, invece, il casting non è richiesto poiché tutti i dati sono già in formato **int**. Non cullatevi, però, in una falsa illusione di sicurezza: se moltiplicate due valori **int** sufficientemente grandi, il risultato potrebbe diventare eccessivo per il tipo di dato, un effetto noto con il famigerato termine di *overflow*. Ecco un esempio che illustra questo problema:

```

//: operators/Overflow.java
// Sorpresa! L'overflow in Java!

public class Overflow {
    public static void main(String[] args) {
        int big = Integer.MAX_VALUE;
        System.out.println("big = " + big);
        int bigger = big * 4;
        System.out.println("bigger = " + bigger);
    }
} /* Output:
big = 2147483647
bigger = -4
*///:~

```

Non otterrete alcun errore o avvertimento da parte del compilatore, né eccezioni in fase di esecuzione.

Java è un ottimo linguaggio, ma questo comportamento non lo è affatto.

Le assegnazioni composte non richiedono cast per i valori **char**, **byte** e **short**, anche se eseguono promozioni che producono risultati identici a quelli ottenibili con le operazioni aritmetiche dirette. D'altra parte, la mancanza di cast semplifica certamente il codice.

Notate che, con l'eccezione del tipo **boolean**, qualsiasi tipo primitivo può essere oggetto di casting in un altro tipo primitivo. Di nuovo, quando eseguite un cast in un tipo più piccolo ricordate di tenere conto dell'effetto di una conversione con riduzione; in caso contrario rischiereste una perdita di informazioni.

Esercizio 14 (3) Scrivete un metodo che accetta due argomenti di tipo **String**, utilizza tutti i confronti booleani per raffrontare le due stringhe e ne visualizza i risultati. Per gli operatori **==** e **!=** eseguite anche il test **equals()**. In **main()**, chiamate il vostro metodo con alcuni oggetti **String** diversi.

Riepilogo

Se avete esperienza con qualsiasi linguaggio che adotta una sintassi analoga al C, non vi sarà sfuggita l'analogia pressoché totale degli operatori Java, che non richiedono quasi alcuno sforzo di apprendimento. Diversamente, se ritenete che il capitolo sia troppo impegnativo è opportuno che consultiate la presentazione multimediale *Thinking in C*, disponibile all'indirizzo www.mindview.net.

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

Capitolo 4

Il controllo dell'esecuzione



Come una creatura senziente, un programma deve manipolare il proprio ambiente e operare le proprie scelte durante l'esecuzione. In Java, per compiere scelte si ricorre alle istruzioni per il controllo dell'esecuzione.

Java utilizza tutte le istruzioni per il controllo dell'esecuzione tipiche del linguaggio C, pertanto se avete familiarità con C o C++ quasi tutto quello che leggerete vi sarà familiare. La maggior parte dei linguaggi di programmazione procedurali possiede qualche genere di istruzioni di controllo, e spesso esiste una certa sovrapposizione tra i vari linguaggi. In Java le parole chiave includono **if-else**, **while**, **do-while**, **for**, **return**, **break** e un'istruzione di selezione chiamata **switch**. Tuttavia Java non supporta il tanto bistrattato **goto**, malgrado tale istruzione possa ancora essere il modo più pratico per risolvere determinati problemi: è ancora possibile eseguire un salto di tipo **goto**, ma è molto più vincolato della versione classica di questa istruzione.

true e false

Per determinare il percorso di esecuzione, tutte le istruzioni condizionali si basano sulla verità o falsità di un'espressione condizionale. Un esempio di espressione di questo tipo è **a == b**, che utilizza l'operatore condizionale **==** per verificare se il

valore di **a** equivale al valore di **b**: l'espressione restituisce **true** (vero) o **false** (falso). Tutti gli operatori relazionali esaminati nel capitolo precedente possono essere utilizzati per creare un'istruzione condizionale. Notate che Java non consente l'utilizzo di un numero come **boolean**, a differenza di C e C++, nei quali un valore diverso da zero rappresenta vero, e un valore pari a zero corrisponde a falso. Se desiderate utilizzare un valore non booleano in una verifica booleana, come in **if(a)**, dovete per prima cosa convertire il valore in booleano ricorrendo a un'espressione condizionale, quale **if(a != 0)**.

if-else

La dichiarazione **if-else** è il meccanismo principale per controllare il flusso di un programma. L'istruzione **else** è facoltativa, perciò potete usare **if** con le due sintassi seguenti:

```
if(espressione booleana)
    istruzione
```

oppure

```
if(espressione booleana)
    istruzione
else
    istruzione
```

L'*espressione booleana* deve produrre un risultato di tipo **boolean**; *istruzione* può essere un'istruzione semplice vera e propria, terminata da un punto e virgola, oppure una dichiarazione composta, vale a dire un gruppo di istruzioni semplici raggruppate tra parentesi graffe. In questo libro, ogni volta che si fa riferimento al termine "istruzione" è implicito che essa possa essere semplice o composta. Come esempio dell'istruzione **if-else** di seguito è riportato un metodo **test()**, che vi dirà se un numero è superiore, inferiore o equivalente a un numero di destinazione:

```
//: control/IfElse.java
import static net.mindview.util.Print.*;
public class IfElse {
    static int result = 0;
```

```
static void test(int testval, int target) {
    if(testval > target)
        result = +1;
    else if(testval < target)
        result = -1;
    else
        result = 0; // I valori sono equivalenti
}
public static void main(String[] args) {
    test(10, 5);
    print(result);
    test(5, 10);
    print(result);
    test(5, 5);
    print(result);
}
} /* Output:
1
-1
0
*///:~
```

All'interno del metodo **test()** trovate anche un'istruzione **else if**: non si tratta di una nuova parola chiave, bensì semplicemente di un **else** seguito da una nuova istruzione **if**. Benché Java, come il C / C++, sia un linguaggio "in formato libero", per convenzione si fa rientrare il testo del corpo di un'istruzione di controllo affinché il lettore possa individuare a colpo d'occhio l'inizio e il termine del blocco.

Iterazione

I cicli sono controllati tramite le istruzioni **while**, **do-while** e **for**, talvolta classificate come istruzioni d'iterazione. Un'istruzione si ripete fino a quando l'*espressione booleana* che esegue il controllo restituisce **false**. La sintassi del ciclo **while** è la seguente:

```
while(espressione booleana)
    istruzione
```



L'espressione booleana viene valutata una volta all'inizio del ciclo e prima di ogni successiva ripetizione dell'istruzione. Di seguito è mostrato un esempio che genera numeri casuali fino a quando non viene verificata una particolare condizione:

```
//: control/WhileTest.java
// Dimostrazione del ciclo while.

public class WhileTest {
    static boolean condition() {
        boolean result = Math.random() < 0.99;
        System.out.print(result + ", ");
        return result;
    }

    public static void main(String[] args) {
        while(condition())
            System.out.println("Inside 'while'");
        System.out.println("Exited 'while'");
    }
} /* (Da eseguire per visualizzare l'output) */://:~
```

Il metodo **condition()** utilizza il metodo statico **random()** nella libreria **Math**, che genera un valore **double** compreso tra 0 e 1 (include 0 ma non 1). Il valore **result** si ottiene dall'operatore di confronto **<**, che produce un risultato **boolean**. Quando visualizzate un valore booleano ottenete automaticamente la stringa "true" o "false" appropriata. L'espressione condizionale del ciclo **while** esprime il concetto: "ripeti le istruzioni presenti nel corpo finché **condition()** restituisce **true**".

do-while

La sintassi del ciclo **do-while** è la seguente:

```
do
    istruzione
while(espressione booleana);
```



L'unica differenza tra **while** e **do-while** è nel fatto che l'istruzione presente in **do-while** viene sempre eseguita almeno una volta, anche se l'espressione valuta **false** la prima iterazione. In un **while**, invece, se la verifica condizionale è **false** nella prima iterazione l'istruzione non viene mai eseguita. Nella pratica, il ciclo **do-while** è meno comune di **while**.

for

Il ciclo **for** è probabilmente la forma d'iterazione più spesso utilizzata. Prima di eseguire la prima iterazione, questo ciclo esegue l'inizializzazione; poi esegue la verifica condizionale, e al termine di ogni iterazione un tipo di "passo": incremento o decremento. La sintassi del ciclo **for** è la seguente:

```
for(inizializzazione; espressione booleana; passo)
    istruzione
```

Una qualsiasi delle espressioni relative a *inizializzazione*, *espressione booleana* e *passo* può essere vuota. L'espressione viene valutata prima di ogni iterazione; non appena restituisce **false**, l'esecuzione prosegue dalla riga successiva al ciclo **for**. Al termine di ogni ciclo viene eseguita l'espressione passo.

Generalmente il ciclo **for** è utilizzato per "contare" le operazioni da compiere:

```
//: control/ListCharacters.java
// Dimostra il ciclo "for" elencando
// tutte le lettere ASCII minuscole.
```

```
public class ListCharacters {
    public static void main(String[] args) {
        for(char c = 0; c < 128; c++)
            if(Character.isLowerCase(c))
                System.out.println("value: " + (int)c +
                    " character: " + c);
    }
} /* Output:
value: 97 character: a
value: 98 character: b
value: 99 character: c
value: 100 character: d
```

```

value: 101 character: e
value: 102 character: f
value: 103 character: g
value: 104 character: h
value: 105 character: i
value: 106 character: j
...
*///:~

```

Noteate che la variabile **c** è definita nel punto in cui viene utilizzata, all'interno dell'espressione di controllo del ciclo **for**, anziché all'inizio del metodo **main()**. L'ambito di visibilità di **c** è l'istruzione controllata dal ciclo **for**.

Anche questo programma si serve della classe **java.lang.Character**. La classe "wrapper" **Character** non soltanto ingloba il tipo primitivo **char** in un oggetto, ma fornisce anche metodi di utilità. In questo caso viene applicato il metodo statico **isLowerCase()** di questa classe per verificare se il carattere corrente è una lettera minuscola.

I linguaggi procedurali tradizionali analoghi al C richiedono che tutte le variabili vengano definite all'inizio di un blocco, in modo che quando il compilatore crea un blocco possa anche allocare lo spazio necessario per le variabili. In Java e C++, invece, potete dichiarare le variabili in un punto qualsiasi del blocco, definendole dove vi occorrono. Questo permette uno stile di codifica più naturale e rende il codice più semplice.

Esercizio 1 (1) Scrivete un programma che visualizza i valori da 1 a 100.

Esercizio 2 (2) Scrivete un programma che generi 25 numeri casuali di tipo **int**. Per ogni valore utilizzate un'istruzione **if-else**, per classificarlo come maggiore di, minore di, o uguale a un secondo valore generato casualmente.

Esercizio 3 (1) Modificate l'Esercizio 2 in modo che il codice venga racchiuso in un cosiddetto "ciclo infinito", che rimane in esecuzione finché non viene interrotto da tastiera, di norma tramite la combinazione Ctrl+C.

Esercizio 4 (3) Scrivete un programma che utilizza due cicli **for** nidificati e l'operatore modulo (%) per identificare e visualizzare i numeri primi, vale a dire i numeri interi divisibili soltanto per uno e per se stessi.

Esercizio 5 (4) Ripetete l'Esercizio 10 del capitolo precedente utilizzando l'operatore ternario e un confronto bit a bit per mostrare i valori uno e zero, invece di **Integer.toBinaryString()**.

L'operatore virgola

L'operatore virgola, da non confondere con il separatore virgola utilizzato per separare le definizioni e gli argomenti dei metodi, ha solo un impiego in Java: nell'espressione di controllo di un ciclo **for**. Infatti, sia nell'inizializzazione sia nei passi dell'espressione di controllo possono essere presenti molte istruzioni, separate da virgolette; tali istruzioni vengono valutate consecutivamente.

Mediante l'operatore virgola potete definire diverse variabili all'interno di un ciclo **for**, purché siano dello stesso tipo:

```

//: control/CommaOperator.java

public class CommaOperator {
    public static void main(String[] args) {
        for(int i = 1, j = i + 10; i < 5; i++, j = i * 2) {
            System.out.println("i = " + i + " j = " + j);
        }
    }
} /* Output:
i = 1 j = 11
i = 2 j = 4
i = 3 j = 6
i = 4 j = 8
*///:~

```

La definizione **int** nell'istruzione **for** contiene le variabili **i** e **j**. La porzione d'inizializzazione può contenere un numero arbitrario di definizioni di un solo tipo. La possibilità di definire variabili in un'espressione di controllo è limitata al ciclo **for**: non è infatti possibile adottare questo approccio con altre istruzioni di selezione o iterazione.

Potete notare che, sia nell'inizializzazione sia nei passi, le istruzioni vengono elaborate in sequenza.



La sintassi **foreach**

Per il ciclo **for** Java SE5 ha introdotto una sintassi nuova e più sintetica da utilizzare con array e contenitori, due argomenti che approfondirete più avanti. Si tratta della cosiddetta *sintassi foreach*, e prevede che non sia necessario creare un **int** per contare una sequenza di voci: **foreach** si incarica di produrre ogni voce automaticamente. Per esempio, supponete di avere un array di **float** e di volere selezionare ogni elemento contenuto nell'array:

```
//: control/ForEachFloat.java
import java.util.*;

public class ForEachFloat {
    public static void main(String[] args) {
        Random rand = new Random(47);
        float f[] = new float[10];
        for(int i = 0; i < 10; i++)
            f[i] = rand.nextFloat();
        for(float x : f)
            System.out.println(x);
    }
} /* Output:
0.72711575
0.39982635
0.5309454
0.0534122
0.16020656
0.57799757
0.18847865
0.4170137
0.51660204
0.73734957
*///:~
```

L'array è popolato utilizzando il “vecchio” ciclo **for**, poiché l'accesso ai suoi elementi deve avvenire mediante un indice. Potete vedere la sintassi **foreach** nell'esempio seguente:

```
for(float x : f) {
```



In questo esempio viene definita una variabile **x** di tipo **float**, e ogni elemento di **f** è assegnato in sequenza a **x**.

Qualsiasi metodo restituisca un array è un eccellente candidato all'utilizzo della sintassi **foreach**. Per esempio, la classe **String** dispone del metodo **toCharArray()** che restituisce un array di **char**, e consente di iterare facilmente i caratteri presenti in una stringa:

```
//: control/ForEachString.java

public class ForEachString {
    public static void main(String[] args) {
        for(char c : "An African Swallow".toCharArray())
            System.out.print(c + " ");
    }
} /* Output:
A n   A f r i c a n   S w a l l o w
*///:~
```

Come vedrete nel Capitolo 11, Come contenere gli oggetti, **foreach** funziona anche con qualsiasi oggetto che implementi l'interfaccia **Iterable**.

Molte istruzioni **for** implicano l'iterazione di una sequenza di valori interi, come in questo esempio:

```
for(int i = 0; i < 100; i++)
```

In casi come questo la sintassi **foreach** non funzionerà, a meno di non creare prima un array di **int**. Per semplificare questo compito **net.mindview.util.Range** offre il metodo **range()**, che genera in modo automatico l'array appropriato. Lo scopo è utilizzare **range()** come importazione **static**:

```
//: control/ForEachInt.java
import static net.mindview.util.Range.*;
import static net.mindview.util.Print.*;

public class ForEachInt {
    public static void main(String[] args) {
        for(int i : range(10)) // 0..9
            printnb(i + " ");
    }
}
```



```

print();
for(int i : range(5, 10)) // 5..9
    printnb(i + " ");
print();
for(int i : range(5, 20, 3)) // 5..20 passo 3
    printnb(i + " ");
print();
}
} /* Output:
0 1 2 3 4 5 6 7 8 9
5 6 7 8 9
5 8 11 14 17
*//*:~
```

Il metodo **range()** è stato “sovraffunzionato” in modo che lo stesso nome di metodo sia utilizzabile con vari elenchi di argomenti; apprenderete presto il funzionamento di questo meccanismo, noto anche come *overloading*. La prima forma sovraffunziona di **range()** inizia da zero e produce valori fino al limite superiore dell’intervallo passato come argomento, meno uno. La seconda forma accetta due argomenti: inizia al valore fornito come primo argomento, e termina al valore fornito come secondo argomento, meno uno. La terza forma, infine, accetta un terzo argomento (*step*) che indica il valore dell’incremento da applicare a ogni iterazione. Come vedrete nel prosieguo del manuale, **range()** è una versione molto semplice di ciò che viene chiamato *generatore*.

Tenete presente che, benché **range()** permetta l’utilizzo della sintassi **foreach** in un numero maggiore di situazioni, aumentando così la leggibilità del codice, è meno efficiente. Se ritenete che le prestazioni siano una caratteristica irrinunciabile potreste utilizzare un *profiler*, ovvero uno strumento che valuta le prestazioni del vostro codice.

Non vi sarà sfuggito l’utilizzo di **printnb()**: questo metodo, che non produce il carattere di nuova riga (*newline*), permette di assemblare le parti che compongono la riga.

La sintassi **foreach** non ha soltanto il vantaggio di essere più sintetica, ma soprattutto è di gran lunga più facile da leggere e indica che cosa state cercando di fare (ottenere ogni elemento dell’array), invece di fornire i dettagli sul modo per farlo (creare un indice da utilizzare per la selezione di ciascun elemento dell’array). Tenete presente che in questo libro la sintassi **foreach** verrà utilizzata ogni volta che sia possibile.



return

Alcune parole chiave rappresentano il cosiddetto *salto incondizionato* (*unconditional branching*), un’espressione che indica semplicemente che la direzione del flusso di esecuzione viene presa senza alcuna verifica: tra queste parole chiave figurano **return**, **break**, **continue** e un meccanismo per saltare a una istruzione fornita di etichetta, simile al **goto** disponibile in altri linguaggi.

La parola chiave **return** ha due funzioni: specifica quale valore viene restituito da un metodo, se questo è privo di un valore di ritorno **void**, e provoca l’uscita dal metodo corrente, restituendo il valore suddetto. Il metodo **test()** visto in precedenza può essere riscritto per sfruttare questo meccanismo:

```

//: control/IfElse2.java
import static net.mindview.util.Print.*;

public class IfElse2 {
    static int test(int testval, int target) {
        if(testval > target)
            return +1;
        else if(testval < target)
            return -1;
        else
            return 0; // Corrispondenza
    }
    public static void main(String[] args) {
        print(test(10, 5));
        print(test(5, 10));
        print(test(5, 5));
    }
} /* Output:
1
-1
0
*//*:~
```

Osservate l’assenza di **else**, inutile poiché il metodo non proseguirà dopo avere eseguito un **return**.

Se non avete un'istruzione **return** in un metodo che restituisce **void**, al termine del metodo stesso esiste comunque un **return** implicito; quindi non è sempre indispensabile includere questa istruzione. Tuttavia, se il metodo dichiara di restituire qualsiasi cosa che non sia **void**, dovete assicurarvi che ogni percorso del codice restituisca un valore.

Esercizio 6 (2) Modificate i metodi **test()** nei due programmi precedenti in modo che accettino due argomenti supplementari, **begin** ed **end**, e che **testval** venga valutato per verificare se si trova nell'intervallo tra **begin** ed **end** inclusi.

break e continue

Potete anche controllare il flusso di un ciclo dall'interno del corpo di qualsiasi istruzione di iterazione, utilizzando **break** e **continue**: **break** interrompe il ciclo senza eseguire le rimanenti istruzioni in esso contenute, mentre **continue** interrompe l'esecuzione dell'iterazione corrente e ritorna all'inizio del ciclo per iniziare l'iterazione successiva.

Questo programma mostra esempi di **break** e **continue** nei cicli **for** e **while**:

```
//: control/BreakAndContinue.java
// Dimostra l'uso delle parole chiave break e continue.
import static net.mindview.util.Range.*;

public class BreakAndContinue {
    public static void main(String[] args) {
        for(int i = 0; i < 100; i++) {
            if(i == 74) break; // Uscita dal ciclo for
            if(i % 9 != 0) continue; // Iterazione successiva
            System.out.print(i + " ");
        }
        System.out.println();
        // Uso di foreach:
        for(int i : range(100)) {
            if(i == 74) break; // Uscita dal ciclo for
            if(i % 9 != 0) continue; // Iterazione successiva
            System.out.print(i + " ");
        }
    }
}
```

```
System.out.println();
int i = 0;
// Un "ciclo infinito":
while(true) {
    i++;
    int j = i * 27;
    if(j == 1269) break; // Uscita dal ciclo for
    if(i % 10 != 0) continue; // Iterazione successiva
    System.out.print(i + " ");
}
} /* Output:
0 9 18 27 36 45 54 63 72
0 9 18 27 36 45 54 63 72
10 20 30 40
*///:~
```

Nel ciclo **for** il valore di **i** non arriva mai a 100, poiché l'istruzione **break** provoca l'uscita dal ciclo non appena **i** vale 74. Di norma utilizzerete un'istruzione **break** come questa soltanto se non sapete quando si verifica la condizione che termina il ciclo.

L'istruzione **continue** fa in modo che l'esecuzione ritorni all'inizio del ciclo, incrementando così **i**, quando il valore di questa variabile non è esattamente divisibile per 9: invece, quando è divisibile per 9, il valore viene visualizzato.

Il secondo ciclo **for** mostra come applicare la sintassi **foreach** per ottenere lo stesso risultato.

In teoria il meccanismo del ciclo **while** “infinito” dovrebbe continuare all'infinito: per evitare questo, nel ciclo è presente un'istruzione **break** che consente l'uscita.

Potete anche vedere che l'istruzione **continue** riporta il controllo all'inizio del ciclo, senza eseguire il codice che segue questa istruzione; pertanto, nel secondo ciclo la visualizzazione avviene soltanto quando il valore di **i** è divisibile per 10. Nell'output viene visualizzato il valore 0 perché l'operazione **0 % 9** restituisce 0.

Una seconda forma di ciclo infinito è **for(;;)**. Il compilatore gestisce **while(true)** e **for(;;)** allo stesso modo; la versione che sceglierete dipende esclusivamente dalle vostre preferenze.

Esercizio 7 (1) Modificate l'Esercizio 1 in modo da uscire dal programma utilizzando la parola chiave **break** al valore 99. In alternativa, proprie a utilizzare anche **return**.

L'“abominevole” **goto**

La parola chiave **goto** è presente già nei primi linguaggi di programmazione. In realtà, **goto** è la genesi del controllo del flusso nei programmi in linguaggio Assembler: “se si verifica la condizione A, salta qui, altrimenti salta là”. Se leggete il codice Assembler generato da qualsiasi compilatore vedrete che il controllo del flusso programmatico contiene molti salti. Anche il compilatore Java produce una sorta di “istruzioni Assembler”, tuttavia questo codice viene eseguito dalla macchina virtuale Java (*Java Virtual Machine*), non direttamente dalla CPU.

Un **goto** è un salto a livello di codice sorgente, ed è a questo che si deve la sua pessima reputazione. Se un programma continua a saltare da un punto all'altro, non vi è modo di strutturare il codice affinché il flusso non sia così irregolare? L'istruzione **goto** è caduta in disgrazia con la pubblicazione del famoso articolo “*Goto considered harmful*” (Il **goto** è da considerarsi dannoso) di Edsger Dijkstra: da allora, bistrattare il **goto** è quasi diventato uno sport.

Come generalmente accade in questo tipo di situazioni, è sempre una questione di equilibrio. Il problema non è tanto l'uso di **goto** quanto l'abuso di questa parola chiave; in sporadiche situazioni **goto** è effettivamente il modo migliore per strutturare il flusso del programma.

Benché in Java **goto** sia una parola riservata, non viene utilizzata nel linguaggio. Java non ha l'istruzione **goto**, tuttavia dispone di un meccanismo che “fa da ponte” tra le parole chiave **break** e **continue**. In realtà non si tratta di un ponte vero e proprio, quanto piuttosto di un modo per interrompere un'iterazione, che viene accomunato al **goto** poiché utilizza la stessa tecnica: un'etichetta.

In questo contesto l'etichetta è un identificativo seguito da due punti (:), come in:

```
label1:
```

In Java, il solo punto in cui è utile collocare un'etichetta è subito prima di un'iterazione. È bene ribadirlo: immediatamente prima; non è opportuno inserire altre istruzioni tra l'etichetta e l'iterazione. E l'unica ragione per inserire un'etichetta prima di un'iterazione è per annidare al suo interno un'al-

tra iterazione o uno **switch**, argomento che sarà trattato tra breve. Questo perché le parole chiave **break** e **continue** normalmente interrompono soltanto il ciclo corrente; invece, se sono utilizzate con un'etichetta interrompono i cicli fino all'etichetta stessa:

```
label1:
    iterazione-esterna {
        iterazione-interna {
            //...
            break; // (1)
            //...
            continue; // (2)
            //...
            continue label1; // (3)
            //...
            break label1; // (4)
        }
    }
```

In (1), **break** esce dall'iterazione interna, per ritrovarsi in quella esterna. In (2), **continue** riporta all'inizio dell'iterazione interna. In (3), invece, l'istruzione **continue label1** interrompe entrambe le iterazioni, riportando il flusso all'inizio (**label1**); di fatto, quindi, lascia proseguire l'iterazione ma inizia da quella esterna. In (4), anche l'istruzione **break label1** interrompe le due iterazioni ritornando a **label1**, ma senza rientrare nel ciclo: di conseguenza esce da entrambe le iterazioni.

Ecco un esempio che applica i cicli **for**:

```
//: control/LabeledFor.java
// Cicli for con break e continue associati a etichetta
import static net.mindview.util.Print.*;

public class LabeledFor {
    public static void main(String[] args) {
        int i = 0;
        outer: // Qui non possono esserci istruzioni
        for(; true ;) { // ciclo infinito
            inner: // Qui non possono esserci istruzioni
            for(; i < 10; i++) {
```



138

```

print("i = " + i);
if(i == 2) {
    print("continue");
    continue;
}
if(i == 3) {
    print("break");
    i++; // Altrimenti i non
          // viene mai incrementata.
    break;
}
if(i == 7) {
    print("continue outer");
    i++; // Altrimenti i non
          // viene mai incrementata.
    continue outer;
}
if(i == 8) {
    print("break outer");
    break outer;
}
for(int k = 0; k < 5; k++) {
    if(k == 3) {
        print("continue inner");
        continue inner;
    }
}
// Qui non possono esserci break o continue che puntano a
// un'etichetta
}
} /* Output:
i = 0
continue inner
i = 1
continue inner

```



139

```

i = 2
continue
i = 3
break
i = 4
continue inner
i = 5
continue inner
i = 6
continue inner
i = 7
continue outer
i = 8
break outer
*//*:~
```

Osservate che **break** esce dal ciclo **for**, e per questo l'espressione di incremento non si verifica fino al termine del passaggio attraverso il ciclo **for**. Dal momento che **break** salta l'espressione di incremento, questo viene eseguito direttamente nel caso di **i == 3**. Anche l'istruzione **continue outer**, nel caso di **i == 7**, va all'inizio del ciclo e salta l'incremento, cosicché viene anche incrementata direttamente.

Se non fosse per l'istruzione **break outer** non vi sarebbe alcun modo per uscire dal ciclo esterno dall'interno del ciclo interno, poiché **break** può uscire soltanto dal ciclo interno: lo stesso vale per **continue**.

Ovviamente, nei casi in cui l'uscita da un ciclo comporti anche l'uscita da un metodo potete semplicemente utilizzare **return**.

L'esempio che segue è una dimostrazione delle istruzioni **break** e **continue** con etichetta, con i cicli **while**:

```

//: control/LabeledWhile.java
// Cicli while con break e continue associati a etichetta
import static net.mindview.util.Print.*;

public class LabeledWhile {
    public static void main(String[] args) {
        int i = 0;
```



```
outer:  
while(true) {  
    print("Outer while loop");  
    while(true) {  
        i++;  
        print("i = " + i);  
        if(i == 1) {  
            print("continue");  
            continue;  
        }  
        if(i == 3) {  
            print("continue outer");  
            continue outer;  
        }  
        if(i == 5) {  
            print("break");  
            break;  
        }  
        if(i == 7) {  
            print("break outer");  
            break outer;  
        }  
    }  
}  
}  
}  
} /* Output:  
Outer while loop  
i = 1  
continue  
i = 2  
i = 3  
continue outer  
Outer while loop  
i = 4  
i = 5  
break
```

```
Outer while loop  
i = 6  
i = 7  
break outer  
*//:/~
```

Le stesse regole valgono per **while**.

1. Un semplice **continue** va all'inizio del ciclo interno e prosegue.
2. Un **continue** con etichetta va all'etichetta e rientra nel ciclo esattamente dopo questa.
3. Un **break** esce dal ciclo.
4. Un **break** con etichetta esce dopo la fine del ciclo indicato dall'etichetta stessa.

È importante ricordare che l'unico motivo per utilizzare le etichette in Java è quando vi sono cicli annidati e si vuole utilizzare **break** o **continue** attraverso più livelli annidati.

Nel suo articolo “Goto considered harmful”, ciò che Dijkstra ha specificamente contestato erano le etichette, non il **goto**. Dijkstra ha osservato che il numero di bug sembra aumentare con il numero di etichette presenti in un programma, e che la presenza di etichette e dell'istruzione **goto** rende i programmi difficili da analizzare. In Java le etichette non danno luogo a questi inconvenienti, poiché la loro posizione è vincolata e non possono essere utilizzate per trasferire il controllo in modo arbitrario. È anche interessante osservare che questo è uno dei pochi casi in cui una funzionalità del linguaggio è resa più utile grazie alla limitazione della potenza dell'istruzione.

switch

L'istruzione **switch**, chiamata talvolta *istruzione di selezione*, consente di scegliere tra porzioni di codice basandosi sul valore di un'espressione intera. La sua sintassi generale è la seguente:

```
switch(selettore-di-intero) {  
    case valore-intero1 : istruzione; break;  
    case valore-intero2 : istruzione; break;  
    case valore-intero3 : istruzione; break;  
    case valore-intero4 : istruzione; break;
```



```
case valore-intero5 : istruzione; break;  
// ...  
default: istruzione;  
}
```

Selettore-di-intero (*integral-selector*) è un'espressione che produce un valore intero.

L'istruzione **switch** confronta il risultato dell'espressione selettore-di-intero con ciascun *valore-intero*: in caso di corrispondenza esegue l'istruzione corrispondente; quest'ultima può essere singola o composta di più istruzioni, e le parentesi graffe non sono necessarie. Se invece non si verifica alcuna corrispondenza, viene eseguita l'istruzione **default**.

Nell'esempio precedente avrete notato che ogni **case** termina con un **break**, che trasferisce l'esecuzione al termine del corpo dell'istruzione **switch**. Per convenzione questo è il modo di costruire un'istruzione **switch**, tuttavia la parola chiave **break** è facoltativa: se assente, il codice delle successive istruzioni **case** viene eseguito finché non si incontra un **break**. Benché di norma questo genere di comportamento non sia desiderabile, potrebbe essere utile a un programmatore esperto. Notate anche che l'ultima istruzione, che segue **default**, è priva di **break** poiché da quel punto l'esecuzione continua comunque dalla riga di codice in cui verrebbe trasferita dal **break**: se lo riteneate importante per questioni di stile, potete inserire un **break** al termine dell'istruzione **default**: in ogni caso il comportamento non cambia.

L'istruzione **switch** è una tecnica elegante per realizzare selezioni multiple tra vari percorsi di esecuzione, ma richiede che il selettore venga valutato come valore intero, per esempio **int** o **char**. Come selettore non è possibile utilizzare, per esempio, una stringa o un numero in virgola mobile, poiché l'istruzione **switch** non funzionerebbe. Per i tipi di dato non interi dovete utilizzare una serie di istruzioni **if**. Alla fine del prossimo capitolo vedrete che la nuova funzionalità **enum**, introdotta da Java SE5, attenua questa limitazione, poiché è stata progettata per funzionare perfettamente con l'istruzione **switch**.

Di seguito è presentato un esempio che genera casualmente alcune lettere e determina se sono vocali, consonanti, o semiconsonanti:

```
//: control/VowelsAndConsonants.java  
// Dimostra il funzionamento dell'istruzione switch.  
import java.util.*;  
import static net.mindview.util.Print.*;
```



```
public class VowelsAndConsonants {  
    public static void main(String[] args) {  
        Random rand = new Random(47);  
        for(int i = 0; i < 100; i++) {  
            int c = rand.nextInt(26) + 'a';  
            printnb((char)c + ", " + c + ": ");  
            switch(c) {  
                case 'a':  
                case 'e':  
                case 'i':  
                case 'o':  
                case 'u': print("vowel");  
                    break;  
                case 'y':  
                case 'w': print("Sometimes a vowel");  
                    break;  
                default: print("consonant");  
            }  
        }  
    }  
} /* Output:  
y, 121: Sometimes a vowel  
n, 110: consonant  
z, 122: consonant  
b, 98: consonant  
r, 114: consonant  
n, 110: consonant  
y, 121: Sometimes a vowel  
g, 103: consonant  
c, 99: consonant  
f, 102: consonant  
o, 111: vowel  
w, 119: Sometimes a vowel  
z, 122: consonant  
...  
*///:~
```



Random.nextInt(26) genera un valore tra 0 e 26, al quale è sufficiente aggiungere il carattere ‘a’ per produrre le lettere minuscole. I caratteri contenuti tra virgolette singole nelle istruzioni **case** producono anche valori interi, utilizzati per eseguire un confronto.

Nelle ultime due coppie di **case**, notate come questi possono essere “accatastati” l’uno sull’altro per fornire riscontri multipli a fronte di una specifica porzione di codice. Ricordate che è essenziale inserire l’istruzione **break** al termine di un determinato **case**, per evitare che l’elaborazione prosegua al **case** successivo.

Nell’istruzione

```
int c = rand.nextInt(26) + 'a';
```

Random.nextInt() produce un valore **int** casuale da 0 a 25, che viene sommato al valore di ‘a’: questo significa che ‘a’ viene automaticamente convertito in **int** per eseguire la somma.

Per visualizzare **c** come carattere è necessario convertirlo in **char**, altrimenti sarebbe visualizzato un numero intero.

Esercizio 8 (2) Create un’istruzione **switch** che visualizza un messaggio per ogni **case** e inserite lo **switch** in un ciclo **for**, per testare ogni **case**. Inserite un **break** dopo ogni **case** e provate il programma, quindi rimuovete le istruzioni **break** e osservate ciò che accade.

Esercizio 9 (4) La serie di Fibonacci è la sequenza dei numeri 1, 1, 2, 3, 5, 8, 13, 21, 34 e così via, nella quale ogni numero, a partire dal terzo, è la somma dei due precedenti.

Create un metodo che accetta un valore intero come argomento e visualizza una quantità equivalente di numeri di Fibonacci a partire dall’inizio della serie: per esempio, se eseguite **java Fibonacci 5**, dove **Fibonacci** è il nome della classe, il prodotto sarà 1, 1, 2, 3, 5.

Esercizio 10 (5) Un “numero vampiro” (*vampire number*) ha un numero pari di cifre, e viene calcolato moltiplicando una coppia di numeri che contengono ciascuna metà delle cifre che compongono il risultato; le cifre sono prese dal numero originale in qualsiasi ordine, e le coppie di zeri finali non sono concesse. Per esempio:

```
1260 = 21 * 60  
1827 = 21 * 87  
2187 = 27 * 81
```



Scrivete un programma che identifica tutti i numeri vampiro di 4 cifre. (Questo esercizio è stato suggerito da Dan Forhan).¹

Riepilogo

Questo capitolo conclude l’analisi delle caratteristiche fondamentali della maggior parte dei linguaggi di programmazione: calcoli, precedenze degli operatori, casting dei tipi, selezione e iterazione. Ora siete pronti a iniziare a compiere i passi che vi avvicineranno al mondo della programmazione a oggetti. Il capitolo successivo tratterà gli importanti argomenti dell’inizializzazione e del cleanup degli oggetti, mentre il capitolo seguente affronterà il concetto essenziale di “occultamento dell’implementazione” (*implementation hiding*).

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

¹. Per la definizione di numero vampiro: <http://groups.google.com/group/sci.math/msg/f17b2281a4aa16da?hl=da&lr=&ie=UTF-8>.



Capitolo 5

Inizializzazione e cleanup



Con il progredire della rivoluzione informatica la programmazione “insicura” è diventata uno dei principali colpevoli dei costi di programmazione elevati.

Due degli argomenti prioritari legati alla sicurezza sono l'*inizializzazione* e il *cleanup*. In C molti bug si verificano poiché i programmatore non inizializzano una variabile: questo si verifica soprattutto con le librerie, quando gli sviluppatori non sanno di dover inizializzare un componente di libreria o semplicemente perché dimenticano di farlo.

Il secondo problema specifico è rappresentato dal *cleanup*: quando si è finito di lavorare con un elemento accade spesso di dimenticarsene, perché “non ci riguarda più”. Così facendo, le risorse utilizzate da quell’elemento rimangono impegnate e possono arrivare a provocare l’esaurimento delle risorse, in particolare della memoria.

C++ ha introdotto il concetto di *costruttore*, un metodo speciale che viene chiamato in modo automatico al momento della creazione di un oggetto. Anche Java ha adottato il costruttore e dispone inoltre di un *garbage collector*, che rilascia automaticamente le risorse di memoria non più in uso. Questo capitolo esamina gli argomenti legati all’inizializzazione e al cleanup e al loro supporto in Java.



L'inizializzazione è garantita dal costruttore

Immaginate di creare un metodo per ogni classe che scrivete, chiamato **initialize()**, un nome che dovrebbe già suggerirvi la necessità di chiamarlo prima di utilizzare l'oggetto in questione: purtroppo, questo significa che dovete ricordarvi di chiamare questo metodo.

In Java il progettista delle classi può assicurare l'inizializzazione di qualsiasi oggetto fornendo un costruttore. Se una classe possiede un costruttore, Java lo chiama automaticamente alla creazione dell'oggetto, ancor prima che l'utente possa accedervi: di fatto, questo comportamento ne garantisce l'inizializzazione.

La sfida successiva consiste nell'assegnare un nome a questo metodo, operazione che comporta due problemi: il primo è che qualsiasi nome utilizziate potrebbe essere in contrasto con un nome che vorreste utilizzare come membro nella classe; il secondo problema è che, dal momento che il compilatore è responsabile della chiamata al costruttore, deve sempre sapere quale metodo chiamare.

La soluzione adottata da C++ sembra essere la più facile e logica, cosicché è stata implementata anche in Java: il nome del costruttore è lo stesso della classe. Il fatto che tale metodo venga chiamato in modo automatico nel corso dell'inizializzazione ha perfettamente senso.

Osservate una semplice classe con un costruttore:

```
//: initialization/SimpleConstructor.java
// Dimostrazione di un semplice costruttore.

class Rock {
    Rock() { // Questo e' il costruttore
        System.out.print("Rock");
    }
}

public class SimpleConstructor {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++)
            new Rock();
    } /* Output:
        Rock Rock Rock Rock Rock Rock Rock Rock Rock
    */
}
```



Rock Rock Rock Rock Rock Rock Rock Rock Rock

*///:~

Ora, quando create un oggetto:

```
new Rock();
```

Java alloca lo spazio necessario in memoria e chiama il costruttore, assicurando in tal modo che l'oggetto sia inizializzato correttamente prima che possiate utilizzarlo.

Noteate che lo stile di codifica secondo il quale la prima lettera dei metodi è minuscola non si applica ai costruttori, poiché il loro nome deve corrispondere esattamente al nome della classe.

Un costruttore che non accetta argomenti è chiamato "costruttore predefinito": la documentazione Java ricorre generalmente all'espressione *no-arg constructor*, tuttavia il concetto di costruttore predefinito è in uso da così tanto tempo prima della comparsa di Java che si è scelto di adottare questo termine. Al pari di qualsiasi altro metodo il costruttore può accettare anche argomenti, che vi consentono di specificare come creare un oggetto. L'esempio precedente può quindi essere modificato facilmente in modo che il costruttore accetti un argomento:

```
//: initialization/SimpleConstructor2.java
// I costruttori possono accettare argomenti.

class Rock2 {
    Rock2(int i) {
        System.out.print("Rock " + i + " ");
    }
}

public class SimpleConstructor2 {
    public static void main(String[] args) {
        for(int i = 0; i < 8; i++)
            new Rock2(i);
    }
} /* Output:
    Rock 0 Rock 1 Rock 2 Rock 3 Rock 4 Rock 5 Rock 6 Rock 7
*/

```

Passando opportuni argomenti al costruttore potete fornirgli i parametri per l'inizializzazione di un oggetto. Per esempio, se la classe **Tree** ha un costruttore che accetta un singolo valore intero che definisce l'altezza di un albero, potrete creare un oggetto **Tree** come il seguente:

```
Tree t = new Tree(12); // albero di 12 metri
```

Se **Tree(int)** è l'unico costruttore il compilatore non permetterà di creare un oggetto **Tree** in nessun altro modo. I costruttori eliminano una ricca tipologia di problemi e rendono il codice più facile da leggere: nel frammento di codice precedente, per esempio, non si nota una chiamata esplicita a qualche metodo **initialize()**, che sarebbe concettualmente separato dalla creazione dell'oggetto. In Java, creazione e inizializzazione sono concetti unificati: non può esistere uno senza l'altro.

Il costruttore è un tipo di metodo insolito, poiché non restituisce alcun valore: questo è un approccio nettamente diverso dal fornire un valore di ritorno **void**, secondo il quale il metodo non restituisce valori ma consente di fargli restituire qualcosa' altro. Di contro, i costruttori non restituiscono nulla né offrono opzioni che vi consentono di farlo: nonostante l'espressione **new** restituisca un riferimento all'oggetto appena creato, il costruttore in sé non ha alcun valore di ritorno. Se fosse presente un valore di ritorno e se poteste impostarne uno voi stessi, il compilatore avrebbe bisogno di sapere, in qualche modo, che cosa fare del valore restituito.

Esercizio 1 (1) Create una classe che contiene un riferimento **String** non inizializzato e dimostrate che Java inizializza automaticamente questo riferimento a **null**.

Esercizio 2 (2) Create una classe con un campo **String** inizializzato al momento della definizione e un altro inizializzato dal costruttore. Qual è la differenza tra le due tecniche?

Overloading dei metodi

Una delle caratteristiche più importanti in ogni linguaggio di programmazione è l'utilizzo dei nomi. Quando create un oggetto, in realtà assegnate un nome a un'area di archiviazione in memoria. Un metodo è il nome di un'azione, e a tutti gli oggetti e metodi si fa riferimento per mezzo di nomi. Con una scelta oculata dei nomi sarete in grado di creare un sistema più facile da comprendere e cambiare; il concetto è molto simile alla scrittura in prosa: l'obiettivo è comunicare con i vostri lettori.

Tuttavia si verifica un problema, quando si cerca di applicare il concetto di "sfumatura" del linguaggio umano anche ai linguaggi di programmazione. Spesso la stessa parola esprime significati diversi, ed è "sovraffabbricato di significati", cioè *overloaded*. Questo è utile, specialmente per esprimere differenze superficiali: nel linguaggio corrente si è soliti dire "lava la camicia", "lava l'automobile" e "lava il cane"; sarebbe ridicolo dover dire invece "camiciaLava la camicia", "automobileLava l'automobile" e "caneLava il cane", soltanto perché l'ascoltatore non debba compiere sforzi mentali per distinguere l'azione effettivamente eseguita. Molti linguaggi umani sono ridondanti, e anche se si perdono alcune parole è sempre possibile ricostruire il significato: non avete bisogno di ricorrere a identificativi poiché il significato è deducibile dal contesto.

Quasi tutti i linguaggi di programmazione, tra cui C, vi chiedono invece di fornire un identificativo univoco per ogni metodo; in questi linguaggi i metodi vengono spesso chiamati *funzioni*. Così facendo, tuttavia, non sarebbe possibile avere una funzione chiamata **print()** per la visualizzazione di numeri interi e un'altra, chiamata anch'essa **print()**, che visualizza i numeri a virgola mobile, dal momento che ciascuna funzione richiede un nome univoco.

In Java e C++, la necessità di metodi sovraccaricati è giustificata da un altro fattore: il costruttore. Visto che il nome del costruttore è predeterminato in base al nome della classe, in teoria può esistere un solo costruttore. Ma se desiderate avere a disposizione più modi per inizializzare un oggetto? Per esempio, supponete di voler costruire una classe inizializzabile sia in modo normale sia leggendo opportune informazioni da un file; in tal caso avreste bisogno di due costruttori: quello predefinito, e un metodo che accetta come argomento una stringa che rappresenta il nome del file con cui inizializzare l'oggetto. Entrambi sono costruttori, quindi entrambi devono avere lo stesso nome della classe. Per questo motivo l'*overloading dei metodi* si rivela fondamentale per utilizzare lo stesso nome di metodo con argomenti di tipo diverso. Benché il sovraccarico dei metodi sia basilare per i costruttori, rimane in ogni caso una funzionalità di uso generale che può essere impiegata con qualsiasi metodo.

Di seguito è riportato un esempio che illustra metodi e costruttori sovraccaricati.

```
//: initialization/Overloading.java
// Dimostrazione che i costruttori e i metodi
// normali possono essere sovraccaricati.
import static net.mindview.util.Print.*;
```

```

class Tree {
    int height;
    Tree() {
        print("Planting a seedling");
        height = 0;
    }
    Tree(int initialHeight) {
        height = initialHeight;
        print("Creating new Tree that is " +
            height + " feet tall");
    }
    void info() {
        print("Tree is " + height + " feet tall");
    }
    void info(String s) {
        print(s + ": Tree is " + height + " feet tall");
    }
}

public class Overloading {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
            Tree t = new Tree(i);
            t.info();
            t.info("overloaded method");
        }
        // Costruttore sovraccarico:
        new Tree();
    }
} /* Output:
Creating new Tree that is 0 feet tall
Tree is 0 feet tall
overloaded method: Tree is 0 feet tall
Creating new Tree that is 1 feet tall
Tree is 1 feet tall
overloaded method: Tree is 1 feet tall

```

```

Creating new Tree that is 2 feet tall
Tree is 2 feet tall
overloaded method: Tree is 2 feet tall
Creating new Tree that is 3 feet tall
Tree is 3 feet tall
overloaded method: Tree is 3 feet tall
Creating new Tree that is 4 feet tall
Tree is 4 feet tall
overloaded method: Tree is 4 feet tall
Planting a seedling///:~

```

Come un vero albero, un oggetto **Tree** può essere creato piantando un seme, senza argomenti, oppure come una giovane pianta coltivata in serra, con un'altezza definita. Per supportare queste possibilità sono presenti un costruttore predefinito e uno che accetta l'altezza da impostare.

Potreste anche volere chiamare il metodo **info()** in modi diversi: per esempio, al fine di visualizzare un messaggio aggiuntivo potreste voler utilizzare **info(String)**, e **info()** per non visualizzare nulla. Sarebbe tuttavia strano assegnare due nomi diversi a qualcosa che rappresenta evidentemente lo stesso concetto. Per fortuna, il sovraccarico dei metodi consente di utilizzare lo stesso nome per entrambi i metodi.

Come distinguere i metodi overloaded

Se i metodi hanno lo stesso nome, come può Java distinguere quale volete usare? La risposta è semplice: ogni metodo sovraccarico deve accettare un elenco univoco di tipi di argomento.

Riflettendoci un istante, questo requisito ha effettivamente senso. Che cosa altro potrebbe segnalare a un programmatore la differenza tra due metodi omonimi, se non i tipi dei loro argomenti? Certo, anche le differenze nell'ordine di argomenti sono sufficienti per distinguere due metodi, tuttavia nella pratica questo approccio non viene adottato spesso dal momento che genera codice difficile da gestire:

```

//: initialization/OverloadingOrder.java
// Overloading basato sull'ordine degli argomenti.
import static net.mindview.util.Print.*;

public class OverloadingOrder {

```

```

static void f(String s, int i) {
    print("String: " + s + ", int: " + i);
}
static void f(int i, String s) {
    print("int: " + i + ", String: " + s);
}
public static void main(String[] args) {
    f("String first", 11);
    f(99, "Int first");
}
} /* Output:
String: String first, int: 11
int: 99, String: Int first
*/

```

I due metodi **f()** hanno argomenti identici, ma il loro ordine è diverso e questo li rende distinti.

Overloading con primitivi

Un tipo primitivo può essere promosso automaticamente da un tipo più piccolo a uno più grande, una funzionalità che può essere in qualche modo fuorviante se combinata con la tecnica di overloading. L'esempio che segue dimostra che cosa accade passando un primitivo a un metodo sovraccarico:

```

//: initialization/PrimitiveOverloading.java
// Promozione di tipi primitivi e overloading.
import static net.mindview.util.Print.*;

public class PrimitiveOverloading {
    void f1(char x) { printnb("f1(char) "); }
    void f1(byte x) { printnb("f1(byte) "); }
    void f1(short x) { printnb("f1(short) "); }
    void f1(int x) { printnb("f1(int) "); }
    void f1(long x) { printnb("f1(long) "); }
    void f1(float x) { printnb("f1(float) "); }
    void f1(double x) { printnb("f1(double) "); }
}

```

```

void f2(byte x) { printnb("f2(byte) "); }
void f2(short x) { printnb("f2(short) "); }
void f2(int x) { printnb("f2(int) "); }
void f2(long x) { printnb("f2(long) "); }
void f2(float x) { printnb("f2(float) "); }
void f2(double x) { printnb("f2(double) "); }

void f3(short x) { printnb("f3(short) "); }
void f3(int x) { printnb("f3(int) "); }
void f3(long x) { printnb("f3(long) "); }
void f3(float x) { printnb("f3(float) "); }
void f3(double x) { printnb("f3(double) "); }

void f4(int x) { printnb("f4(int) "); }
void f4(long x) { printnb("f4(long) "); }
void f4(float x) { printnb("f4(float) "); }
void f4(double x) { printnb("f4(double) "); }

void f5(long x) { printnb("f5(long) "); }
void f5(float x) { printnb("f5(float) "); }
void f5(double x) { printnb("f5(double) "); }

void f6(float x) { printnb("f6(float) "); }
void f6(double x) { printnb("f6(double) "); }

void f7(double x) { printnb("f7(double) "); }

void testConstVal() {
    printnb("5: ");
    f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5); print();
}

void testChar() {
    char x = 'x';
    printnb("char: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
}

```



```

void testByte() {
    byte x = 0;
    println("byte: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
}
void testShort() {
    short x = 0;
    println("short: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
}
void testInt() {
    int x = 0;
    println("int: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
}
void testLong() {
    long x = 0;
    println("long: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
}
void testFloat() {
    float x = 0;
    println("float: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
}
void testDouble() {
    double x = 0;
    println("double: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
}

public static void main(String[] args) {
    PrimitiveOverloading p =
        new PrimitiveOverloading();
    p.testConstVal();
    p.testChar();
    p.testByte();
}

```



```

    p.testShort();
    p.testInt();
    p.testLong();
    p.testFloat();
    p.testDouble();
}
} /* Output:
5: f1(int) f2(int) f3(int) f4(int) f5(long) f6(float)
f7(double)
char: f1(char) f2(int) f3(int) f4(int) f5(long) f6(float)
f7(double)
byte: f1(byte) f2(byte) f3(short) f4(int) f5(long) f6(float)
f7(double)
short: f1(short) f2(short) f3(short) f4(int) f5(long)
f6(float) f7(double)
int: f1(int) f2(int) f3(int) f4(int) f5(long) f6(float)
f7(double)
long: f1(long) f2(long) f3(long) f4(long) f5(long) f6(float)
f7(double)
float: f1(float) f2(float) f3(float) f4(float) f5(float)
f6(float) f7(double)
double: f1(double) f2(double) f3(double) f4(double) f5(double)
f6(double) f7(double)
*///:~


```

Come potete vedere il valore costante 5 viene trattato come **int**, cosicché Java utilizzerà, se disponibile, un metodo sovraccarico che accetta **int**. In tutti gli altri casi, se esiste un tipo di dato più piccolo di quello dell'argomento del metodo, questo sarà il dato promosso. Il tipo **char** produce un effetto lievemente diverso, in quanto in mancanza di un metodo che accetta un **char** viene promosso a **int**.

Osservate che cosa accade, invece, qualora il tipo di dato dell'argomento sia più grande di quello atteso dal metodo sovraccarico, con una modifica al programma precedente.

```

//: initialization/Demotion.java
// Retrocessione di tipi primitivi e overloading.
import static net.mindview.util.Print.*;

```



```

public class Demotion {
    void f1(char x) { print("f1(char)"); }
    void f1(byte x) { print("f1(byte)"); }
    void f1(short x) { print("f1(short)"); }
    void f1(int x) { print("f1(int)"); }
    void f1(long x) { print("f1(long)"); }
    void f1(float x) { print("f1(float)"); }
    void f1(double x) { print("f1(double)"); }

    void f2(char x) { print("f2(char)"); }
    void f2(byte x) { print("f2(byte)"); }
    void f2(short x) { print("f2(short)"); }
    void f2(int x) { print("f2(int)"); }
    void f2(long x) { print("f2(long)"); }
    void f2(float x) { print("f2(float)"); }

    void f3(char x) { print("f3(char)"); }
    void f3(byte x) { print("f3(byte)"); }
    void f3(short x) { print("f3(short)"); }
    void f3(int x) { print("f3(int)"); }
    void f3(long x) { print("f3(long)"); }

    void f4(char x) { print("f4(char)"); }
    void f4(byte x) { print("f4(byte)"); }
    void f4(short x) { print("f4(short)"); }
    void f4(int x) { print("f4(int)"); }

    void f5(char x) { print("f5(char)"); }
    void f5(byte x) { print("f5(byte)"); }
    void f5(short x) { print("f5(short)"); }

    void f6(char x) { print("f6(char)"); }
    void f6(byte x) { print("f6(byte)"); }

    void f7(char x) { print("f7(char)"); }
}

```



```

void testDouble() {
    double x = 0;
    print("double argument:");
    f1(x);f2((float)x);f3((long)x);f4((int)x);
    f5((short)x);f6((byte)x);f7((char)x);
}

public static void main(String[] args) {
    Demotion p = new Demotion();
    p.testDouble();
}

/* Output:
double argument:
f1(double)
f2(float)
f3(long)
f4(int)
f5(short)
f6(byte)
f7(char)
*//*:~

```

In questo codice i metodi accettano valori primitivi più piccoli. Se il vostro argomento è più grande dovrete eseguire una conversione di tipo *narrowing* ricorrendo a un cast opportuno: diversamente il compilatore visualizzerà un messaggio di errore.

Overloading sui valori di ritorno

Due domande che i neofiti si pongono di frequente sono: "Perché soltanto mediante il nome della classe e l'elenco degli argomenti? Non è possibile distinguere i metodi basandosi sui loro valori di ritorno?". Questi due metodi, per esempio, identici nei nomi e negli argomenti, sono facilmente distinguibili l'uno dall'altro:

```

void f0 {}
int f0 { return 1; }

```

Questa soluzione potrebbe funzionare, purché il compilatore sia in grado di determinare in modo inequivocabile il significato dal contesto, come in `int x = f()`. Tuttavia è anche possibile chiamare un metodo e ignorarne il valore di ritorno, adottando una tecnica nota come “chiamata di un metodo per il suo effetto collaterale”: non è importante il valore restituito, quanto gli altri effetti prodotti dalla chiamata al metodo.

Quindi, chiamando il metodo in questo modo:

```
f();
```

Java non è in grado di determinare quale metodo `f()` deve chiamare. Inoltre, come distinguere i metodi con un semplice esame del codice? Queste ragioni fanno sì che non sia possibile utilizzare il tipo del valore di ritorno per distinguere i metodi sovraccarichi.

Costruttori predefiniti

Come si è detto, un costruttore predefinito è privo di argomenti dal momento che viene impiegato per creare un “oggetto predefinito”. Se create una classe priva di costruttore, il compilatore creerà automaticamente un costruttore predefinito, come nell’esempio seguente:

```
//: initialization/DefaultConstructor.java

class Bird {}

public class DefaultConstructor {
    public static void main(String[] args) {
        Bird b = new Bird(); // Costruttore predefinito!
    }
} ///:~
```

L’espressione

```
new Bird()
```

crea un nuovo oggetto e chiama il costruttore predefinito, anche se non ne è stato creato uno in modo esplicito: senza di esso non avreste alcun metodo

da chiamare per costruire l’oggetto. Tuttavia, se definite un costruttore qualsiasi con o senza argomenti il compilatore non ne creerà uno predefinito:

```
//: initialization/NoSynthesis.java

class Bird2 {
    Bird2(int i) {}
    Bird2(double d) {}
}

public class NoSynthesis {
    public static void main(String[] args) {
        // Bird2 b = new Bird2(); /* Non esiste il costruttore
        // predefinito */

        Bird2 b2 = new Bird2(1);
        Bird2 b3 = new Bird2(1.0);
    }
} ///:~
```

Se scrivete

```
new Bird2()
```

il compilatore farà notare che manca il costruttore adatto. Quando non prevedete un costruttore, è come se il compilatore vi informasse che si occuperà della sua creazione. Se scrivete un costruttore, tuttavia, il compilatore penserà: “avete previsto un costruttore, quindi presumo sappiate ciò che state facendo; non avete introdotto il costruttore predefinito, pertanto ritenete di non averne bisogno”.

Esercizio 3 (1) Create una classe con un costruttore predefinito (che non accetta argomenti) che visualizza un messaggio, e create un oggetto di questa classe.

Esercizio 4 (1) Alla classe precedente aggiungete un costruttore sovraccarico che accetta un argomento `String` e lo visualizza insieme al vostro messaggio.

Esercizio 5 (2) Create una classe chiamata `Dog` con un metodo `bark()` sovraccarico. Questo metodo dovrebbe essere sovraccaricato in base ai diversi tipi di dato primitivo e visualizzare vari tipi di versi del cane:

abbiare, mugolio, guaito ecc., in funzione della versione sovraccarica che verrà chiamata. Scrivete un metodo **main()** per chiamare tutte le versioni dei metodi.

Esercizio 6 (1) Modificate l'esercizio precedente in modo che due dei metodi sovraccarichi accettino argomenti di due tipi diversi, ma in ordine invertito l'uno rispetto all'altro, e verificatene il funzionamento.

Esercizio 7 (1) Create una classe senza costruttore, quindi create un oggetto di quella classe in **main()** per dimostrare che il costruttore predefinito viene creato automaticamente.

Parola chiave this

Se avete due oggetti dello stesso tipo, chiamati **a** e **b**, potreste chiedervi in quale modo sia possibile chiamare un metodo **peel()** per entrambi gli oggetti.

```
//: initialization/BananaPeel.java

class Banana { void peel(int i) { /* ... */ } }

public class BananaPeel {
    public static void main(String[] args) {
        Banana a = new Banana(),
                b = new Banana();
        a.peel(1);
        b.peel(2);
    }
} //://:~
```

Essendoci un solo metodo chiamato **peel()**, come può sapere se la chiamata è per l'oggetto **a** o **b**? Per consentirvi di scrivere codice con una sintassi pratica orientata agli oggetti, mediante la quale “inviate un messaggio a un oggetto”, il compilatore esegue alcune attività a vostra insaputa. In primo luogo passa un argomento segreto al metodo **peel()**, vale a dire il riferimento all'oggetto da manipolare; pertanto le due chiamate di metodo diventano simili a:

```
Banana.peel(a, 1);
Banana.peel(b, 2);
```

Tenete presente che questa è una semplice rappresentazione del comportamento interno, poiché in realtà non è possibile scrivere queste espressioni e forzare il compilatore ad accettarle, tuttavia vi dà un'idea di ciò che avviene.

Supponete di trovarvi all'interno di un metodo e di voler ottenere il riferimento all'oggetto corrente. Poiché questo riferimento è passato segretamente dal compilatore, non possiede identificativo. A questo scopo è tuttavia disponibile la parola chiave **this**: utilizzabile soltanto dall'interno di metodi non **static**, essa restituisce il riferimento all'oggetto per cui è stato chiamato il metodo, vale a dire l'oggetto corrente. Potete considerare questo riferimento esattamente come qualsiasi altro riferimento a oggetti. Ricordate che se chiamate un metodo della vostra classe dall'interno di un altro metodo della stessa non avrete bisogno di utilizzare la parola chiave **this**, e vi basterà semplicemente chiamare il metodo. Il riferimento **this** corrente è utilizzato automaticamente per chiamare l'altro metodo, pertanto potete esprimervi in questo modo:

```
//: initialization/Apricot.java
public class Apricot {
    void pick() { /* ... */ }
    void pit() { pick(); /* ... */ }
} //://:~
```

All'interno di **pit()** potreste scrivere **this.pick()**, ma questo non è indispensabile poiché il compilatore se ne incarica automaticamente.¹

Dovreste servirvi della parola chiave **this** solo nei casi specifici in cui avete necessità di utilizzare esplicitamente il riferimento all'oggetto corrente. Per esempio **this** appare spesso nelle istruzioni **return**, quando si desidera restituire il riferimento all'oggetto corrente:

```
//: initialization/Leaf.java
// Un semplice utilizzo della parola chiave "this".

public class Leaf {
```

1. Alcuni programmati hanno l'osessione di anteporre **this** a ogni chiamata di metodo o riferimento di campo, sostenendo di rendere così il codice “più chiaro ed esplicito”. Non fatelo. Un motivo per utilizzare i linguaggi di alto livello è che essi svolgono il lavoro per noi. Se usate **this** in modo ridondante otterrete di confondere (e annoiare) chiunque legga il vostro codice, considerato che i programmati non fanno un uso smodato di **this**. Questa parola chiave deve essere utilizzata soltanto quando ciò sia indispensabile; inoltre, uno stile di codifica coerente e diretto consentirà di risparmiare tempo e denaro.



```

int i = 0;
Leaf increment() {
    i++;
    return this;
}
void print() {
    System.out.println("i = " + i);
}
public static void main(String[] args) {
    Leaf x = new Leaf();
    x.increment().increment().print();
}
} /* Output:
i = 3
*///:~

```

Considerato che **increment()** restituisce il riferimento dell'oggetto corrente tramite la parola chiave **this**, è possibile eseguire facilmente più operazioni sullo stesso oggetto.

La parola chiave **this** è efficace anche per passare l'oggetto corrente a un altro metodo:

```

//: initialization/PassingThis.java

class Person {
    public void eat(Apple apple) {
        Apple peeled = apple.getPealed();
        System.out.println("Yummy");
    }
}

class Peeler {
    static Apple peel(Apple apple) {
        // ... rimuove la buccia
        return apple; // Sbucciatura eseguita
    }
}

```



```

class Apple {
    Apple getPealed() { return Peeler.peel(this); }
}

public class PassingThis {
    public static void main(String[] args) {
        new Person().eat(new Apple());
    }
} /* Output:
Yummy
*///:~

```

Apple deve chiamare **Peeler.peel()**, un metodo di utilità esterno incaricato di eseguire un'operazione che, per qualche motivo, deve essere esterna ad **Apple**: è possibile, per esempio, che il metodo esterno debba essere applicabile da varie classi e che non vogliate duplicare il codice. Per passare se stesso al metodo esterno, quindi, deve utilizzare **this**.

Esercizio 8 (1) Create una classe con due metodi; all'interno del primo chiamate due volte il secondo, la prima volta senza utilizzare **this** e la seconda con **this**. Tenete presente che questa tecnica ha il solo scopo di esaminare il meccanismo in funzione: nella pratica non dovreste ricorrervi.

Chiamate di costruttori da costruttori

Quando scrivete numerosi costruttori per una classe, potrete incorrere in situazioni in cui vorrete chiamare un costruttore da un altro per evitare di duplicare codice: potete eseguire questa chiamata mediante la parola chiave **this**.

Normalmente quando vi servite di **this** è nel senso di “in questo oggetto” o “nell’oggetto corrente”, allo scopo di produrre il riferimento all’oggetto corrente. In un costruttore, invece, la parola chiave **this** assume un significato diverso quando è associata a un elenco di argomenti, eseguendo una chiamata esplicita al costruttore che corrisponde all’elenco di argomenti in questione. Questa tecnica si rivela molto pratica per chiamare altri costruttori.

```

//: initialization/Flower.java
// Chiamata dei costruttori con "this"
import static net.mindview.util.Print.*;

```



```

public class Flower {
    int petalCount = 0;
    String s = "initial value";
    Flower(int petals) {
        petalCount = petals;
        print("Constructor w/ int arg only, petalCount= "
            + petalCount);
    }
    Flower(String ss) {
        print("Constructor w/ String arg only, s = " + ss);
        s = ss;
    }
    Flower(String s, int petals) {
        this(petals);
    }
    //! this(s); // Non e' possibile chiamarlo due volte!
    this.s = s; // Un altro uso di "this"
    print("String & int args");
}
Flower() {
    this("hi", 47);
    print("default constructor (no args)");
}
void printPetalCount() {
//! this(11); /* Non dall'interno di un metodo
//          non-costruttore*/
    print("petalCount = " + petalCount + " s = " + s);
}
public static void main(String[] args) {
    Flower x = new Flower();
    x.printPetalCount();
}
} /* Output:
Constructor w/ int arg only, petalCount= 47
String & int args
default constructor (no args)
petalCount = 47 s = hi
*///:~

```

Il costruttore **Flower(String s, int petals)** evidenzia che, sebbene sia possibile chiamare un costruttore servendosi di **this**, non è permesso chiamarlo due volte: inoltre, la chiamata al costruttore deve essere la prima operazione eseguita, pena un messaggio di errore del compilatore.

Questo esempio mostra anche un altro modo di utilizzare **this**. Poiché il nome dell'argomento **s** e il nome del dato membro **s** sono identici, si è in presenza di un'ambiguità, che tuttavia può essere risolta utilizzando **this.s** per specificare che si fa riferimento ai dati membro. Nel codice Java vedrete spesso questa forma, anche in numerosi esempi di questo manuale.

Nel metodo **printPetalCount()** potete notare che il compilatore non consente di chiamare un costruttore dall'interno di qualsiasi metodo diverso da un costruttore.

Esercizio 9 (1) Create una classe con due costruttori sovraccarichi. Utilizzando **this** chiamate il secondo costruttore dal primo.

Significato di static

Ora che avete esaminato il funzionamento della parola chiave **this** comprendrete con maggior compiutezza che cosa significa creare un metodo **static**: in sintesi, vuol dire che con il metodo non è possibile usare **this**. Non potete chiamare metodi non **static** dall'interno di metodi **static**, malgrado sia ammesso il contrario.²

Potete anche chiamare un metodo **static** per mezzo della classe stessa, senza che sia necessaria la presenza di un oggetto. In effetti è questo l'obiettivo primario di un metodo statico, concettualmente equivalente alla creazione di un metodo globale. Poiché Java non permette la creazione di metodi globali, l'inserimento di un metodo **static** in una classe consente di accedere ad altri metodi e campi **static**.

Alcuni tecnici fanno notare che i metodi statici non sono strettamente orientati agli oggetti, poiché la loro semantica è tipica dei metodi globali; con un metodo **static** non inviate un messaggio a un oggetto, dal momento che non esiste alcun **this**. Probabilmente questo è un argomento sensato, e se vi trovate a utilizzare molti metodi **static** forse dovreste riconsiderare la vostra strategia.

2. L'unica situazione in cui è permesso farlo è passando un riferimento a un oggetto nel metodo **static**, che può anche creare oggetti propri; quindi, tramite il riferimento (**this**, appunto), è possibile chiamare metodi non statici e accedere a campi non statici. Tenete presente, però, che questa funzionalità viene generalmente ottenuta per mezzo di un normale metodo non **static**.

Comunque sia i metodi **static** sono pratici e in alcuni casi è veramente utile ricorrervi: lasciate ai teorici dell'informatica eventuali considerazioni sulla loro aderenza alla "filosofia OOP".

Cleanup: finalizzazione e garbage collection

I programmati conoscono bene l'importanza dell'inizializzazione, ma spesso dimenticano l'importanza delle operazioni di "pulizia", il cosiddetto *cleanup*. Dopotutto, chi ha bisogno di "ripulire" un **int**? Con l'utilizzo delle librerie, però, limitarsi ad "abbandonare" un oggetto dopo aver terminato di usarlo non è sempre l'approccio migliore. Naturalmente Java mette a disposizione il garbage collector, che recupera la memoria occupata dagli oggetti non più utilizzati. Ora considerate un caso anomalo: supponete che il vostro oggetto allochi memoria "speciale" senza utilizzare **new**. Il garbage collector sa soltanto come rilasciare la memoria assegnata *con new*, non come rilasciare quella "speciale" dell'oggetto. Per gestire questa situazione Java ha un metodo, chiamato **finalize()**, che potete definire nella vostra classe: di seguito è descritto il suo presunto funzionamento. Quando il garbage collector è pronto per rilasciare la memoria utilizzata dal vostro oggetto, per prima cosa chiama il metodo **finalize()** e soltanto nella fase successiva della garbage collection recupera la memoria dell'oggetto. Pertanto, se scegliete di servirvi di **finalize()**, avrete la possibilità di eseguire importanti operazioni di cleanup durante la garbage collection.

Questa caratteristica nasconde un potenziale tranello, poiché alcuni programmati, in particolare quelli di C++, potrebbero confondere il metodo **finalize()** con il *distruttore* di C++, una funzione che viene sempre chiamata al momento della distruzione di un oggetto. In proposito è importante distinguere C++ da Java: in C++, se il programma è privo di errori, gli oggetti vengono sempre distrutti, mentre in Java gli oggetti non sono sempre gestiti dal garbage collector. In altre parole:

1. i vostri oggetti potrebbero non essere soggetti alla garbage collection;
2. la garbage collection non equivale alla "distruzione".

Memorizzando questi due concetti fondamentali non avrete problemi. Il senso di tutto questo è che se volete eseguire qualche operazione prima di abbandonare un oggetto, dovete incaricarvene voi stessi. Java non offre alcun distruttore o funzionalità equivalente, cosicché dovete creare un metodo standard per eseguire il cleanup. Supponete, per esempio, che nel corso del

processo di creazione il vostro oggetto disegni una figura a video: se non cancellate esplicitamente l'immagine essa potrebbe rimanere sullo schermo per un tempo indefinito. Di contro, inserendo una funzionalità di cancellazione in **finalize()**, se l'oggetto sarà preso in carico dal garbage collector e sarà chiamato il metodo **finalize()** (ricordate che non è detto che ciò avvenga), per prima cosa l'immagine sarà rimossa dallo schermo: tuttavia, se questo non accadesse, l'immagine rimarrebbe.

Potrebbe verificarsi il caso che un oggetto non venga mai rilasciato, perché il vostro programma non è mai prossimo a esaurire la memoria a propria disposizione. Tenete presente, in ogni modo, che se il programma termina e il garbage collector non arriva mai a rilasciare la memoria per uno qualunque dei vostri oggetti, al termine del programma la memoria verrà comunque rilasciata al sistema operativo. Questo è un lodevole comportamento, dal momento che la garbage collection ha un costo in termini di gestione: non eseguendola mai, non si incorre in questo onere.

A che cosa serve **finalize()**?

Quindi, se non può essere utilizzato come strumento di cleanup generale, a che cosa serve il metodo **finalize()**?

Un terzo punto da ricordare è il seguente:

3. la garbage collection si occupa soltanto della memoria.

L'unica ragione dell'esistenza del garbage collector, quindi, è il recupero della memoria inutilizzata; di conseguenza qualsiasi attività che sia associata alla garbage collection, in particolare il vostro metodo **finalize()**, dovrà riguardare soltanto la memoria e la sua deallocazione.

Questo non significa che se il vostro oggetto contiene altri oggetti **finalize()** dovrà rilasciarli esplicitamente: il garbage collector si occupa del rilascio dell'intera memoria occupata dagli oggetti, a prescindere da come essi sono stati creati. In realtà, l'utilizzo di **finalize()** è limitato a casi speciali in cui il vostro oggetto può allocare memoria in modo diverso dalla creazione di un oggetto. Tuttavia potreste obiettare che in Java tutto è un oggetto: quindi, come può accadere questo?

Si potrebbe affermare che lo scopo di **finalize()** sia consentire l'esecuzione di operazioni in stile C, allocando memoria tramite meccanismi diversi da quello abitualmente usato in Java. Ciò può avvenire soprattutto attraverso i metodi nativi, che rappresentano un modo per chiamare da Java codice non Java: i metodi nativi sono trattati diffusamente nell'Appendice B della seconda edizione elettronica di questo manuale, disponibile su www.mindview.net.

Al momento C e C++ sono gli unici linguaggi supportati dai metodi nativi, ma poiché è possibile chiamare sottoprogrammi scritti in altri linguaggi, in effetti vi è permesso chiamare qualsiasi programma. Nel codice non Java, l'allocazione di memoria può avvenire chiamando la famiglia di funzioni C **malloc()**; a meno di non chiamare poi **free()**, tale memoria non verrà rilasciata, causando una perdita di memoria (*memory leak*). Ovviamente **free()** è una funzione di C e C++, cosicché potreste avere bisogno di chiamarla in un metodo nativo nel vostro metodo **finalize()**.

Dopo avere letto questo paragrafo, probabilmente non vi verrà voglia di utilizzare spesso **finalize()**.³

In effetti **finalize()** non è il punto ideale in cui eseguire le normali operazioni di cleanup: vi domanderete quindi dove dovrebbero avvenire queste operazioni.

È necessario fare pulizia

Quando lo ritiene necessario, per ripulire la memoria da un oggetto l'utente dovrebbe chiamare un metodo apposito. Questa affermazione è lapalissiana, tuttavia contrasta in qualche modo con il concetto di distruttore in C++. In questo linguaggio tutti gli oggetti vengono distrutti, o per meglio dire tutti gli oggetti dovrebbero esserlo. Se l'oggetto C++ viene creato come oggetto locale, cioè sullo stack (operazione non consentita in Java), la distruzione avviene in corrispondenza della parentesi graffa di chiusura dell'ambito di visibilità in cui l'oggetto è stato creato. Qualora l'oggetto sia stato creato con **new**, come in Java, il distruttore verrà chiamato quando il programmatore chiama l'operatore C++ **delete**, che non esiste in Java. Se il programmatore C++ dimentica di chiamare **delete**, il distruttore non verrà mai chiamato, si avrà una perdita di memoria e le altre parti dell'oggetto non verranno mai eliminate. Questo tipo di bug può essere molto difficile da rintracciare, ed è una delle irresistibili ragioni che invitano a passare a Java.

Di contro Java non consente di creare oggetti locali, rendendo sempre necessario l'utilizzo di **new**. In Java tuttavia non esiste un “delete” che consenta il rilascio dell'oggetto, perché la memoria viene liberata dal garbage collector: quindi, da un punto di vista semplicistico si potrebbe affermare che a causa del garbage collector Java non dispone di un distruttore. Vedrete però nel prosieguo del manuale che la presenza di un garbage collector non annulla la

3. Joshua Bloch, nella sezione intitolata “Avoid finalizers”, è ancora più esplicito: “I metodi **finalize()** sono imprevedibili, spesso pericolosi e di solito inutili.” (*Effective Java™ Programming Language Guide*, p. 20, Addison-Wesley, 2001).

necessità o l'utilità dei distruttori: comunque sia non dovreste mai chiamare **finalize()** direttamente, poiché questa non è una soluzione. Per eseguire in Java operazioni di cleanup diverse dal rilascio della memoria dovete chiamare esplicitamente un metodo apposito, che equivale a un distruttore C++ pur senza presentarne i vantaggi.

Ricordate che non vengono garantite né la garbage collection né la finalizzazione, visto che se la JVM non è prossima all'esaurimento della memoria corrente potrebbe decidere di non perdere tempo attivando il garbage collector.

Condizione di terminazione

In generale non potete contare sul fatto che **finalize()** venga chiamato, quindi dovete creare metodi di cleanup appositi e chiamarli esplicitamente. È quindi evidente che **finalize()** è utile soltanto per oscure operazioni di cleanup della memoria che la maggior parte dei programmatori non eseguirà mai. Tuttavia c'è un uso interessante di **finalize()** che non richiede ogni volta una chiamata a questo metodo: si tratta della cosiddetta verifica della *condizione di terminazione* di un oggetto.⁴

Quando non siete più interessati a un oggetto, esso è pronto per essere cancellato, e dovrebbe quindi trovarsi in uno stato in cui la sua memoria può essere rilasciata in tutta sicurezza: per esempio, un oggetto che rappresenta un file aperto dovrebbe essere chiuso dal programmatore prima di sottoporre l'oggetto alla garbage collection. Qualora una qualsiasi porzione dell'oggetto non venga “ripulita” in modo corretto il vostro programma presenterà un bug, che potrebbe essere molto difficile da individuare. Il metodo **finalize()** può servire a rilevare questa condizione, anche se non sempre viene chiamato. Se una delle operazioni di finalizzazione rileva il bug, avrete scoperto il problema, il che è ciò che vi interessa realmente.

Ecco un semplice esempio dell'impiego di **finalize()**.

```
//: initialization/TerminationCondition.java
// Uso di finalize() per identificare un oggetto
// che non viene correttamente ripulito.
```

```
class Book {
    boolean checkedOut = false;
```

4. “Condizione di terminazione” è un termine coniato da Bill Venners (www.artima.com) nel corso di un seminario tenuto in collaborazione con l'autore.

```

Book(boolean checkOut) {
    checkedOut = checkOut;
}

void checkIn() {
    checkedOut = false;
}

protected void finalize() {
    if(checkedOut)
        System.out.println("Error: checked out");
    // Normalmente eseguireste anche questa operazione:
    // super.finalize(); /* Chiama la versione della
    //                     superclasse*/
}

public class TerminationCondition {
    public static void main(String[] args) {
        Book novel = new Book(true);
        // Cleanup corretto:
        novel.checkIn();
        // Cancella il riferimento, dimenticando il clean up:
        new Book(true);
        // Forza la garbage collection e la finalizzazione:
        System.gc();
    }
}
/* Output:
Error: checked out
*///:~

```

La condizione di terminazione è che tutti gli oggetti **Book** dovrebbero essere controllati prima di essere sottoposti alla garbage collection; tuttavia, in **main()** un errore di programmazione non verifica uno degli oggetti **Book**. Senza l'ausilio di **finalize()** per verificare la condizione di terminazione questo può essere un bug difficile da identificare.

Noteate che per forzare la finalizzazione viene utilizzato il metodo **System.gc()**; anche se così non fosse, comunque, è molto probabile che l'oggetto **Book** errante venga scoperto dopo un certo numero di esecuzioni del pro-

gramma, supponendo che questo allochi memoria sufficiente a far intervenire il garbage collector.

Di norma dovreste supporre che anche la versione di **finalize()** della superclasse esegua qualche operazione importante, e dovreste chiamarla utilizzando **super**, come potete vedere nel metodo **Book.finalize()**: in questo caso specifico la chiamata è stata commentata poiché richiede la gestione delle eccezioni, un argomento che non avete ancora affrontato.

Esercizio 10 (2) Create una classe con un metodo **finalize()** che visualizza un messaggio. In **main()** generare un oggetto della vostra classe e spiegate il comportamento del programma.

Esercizio 11 (4) Modificate l'esercizio precedente in modo che il metodo **finalize()** venga sempre chiamato.

Esercizio 12 (4) Create una classe chiamata **Tank** che possa essere riempita e svuotata, e la cui condizione di terminazione preveda che l'oggetto sia vuoto al momento dell'eliminazione. Scrivete un metodo **finalize()** che verifica tale condizione. In **main()**, inoltre, verificate le possibili ipotesi che possono verificarsi nell'utilizzo di **Tank**.

Funzionamento della garbage collection

Se siete abituati a un linguaggio di programmazione in cui l'allocazione di oggetti sull'heap è un'operazione onerosa, potreste ritenere che lo schema Java secondo il quale tutto viene allocato sull'heap, tranne i tipi primitivi, sia altrettanto impegnativo. Tuttavia, il garbage collector può avere un impatto rilevante nel velocizzare la creazione degli oggetti. A prima vista potrebbe sembrare alquanto strano che il rilascio della memoria ne influenzi l'allocazione, ma è così che funzionano le JVM: ciò lascia intendere che l'allocazione di spazio per gli oggetti nella memoria heap può essere veloce quasi quanto l'allocazione su stack in altri linguaggi.

Per esempio, provate a pensare all'heap di C++ come a un cortile, nel quale ogni oggetto rivendica il proprio angolo di manto erboso; presto o tardi questa proprietà (intesa come bene immobile) può essere abbandonata, e deve essere riutilizzata. In alcune JVM l'heap Java si comporta in modo diverso, più simile a un nastro trasportatore che avanza ogni volta che si assegna un nuovo oggetto. Questo significa che l'assegnazione di memoria per l'oggetto è straordinariamente rapida; il "puntatore all'heap" viene semplicemente spostato in "territorio vergine", comportandosi in modo analogo all'assegnazione su stack in C++: certamente questo comporta un piccolo costo in termini di gestione, ma nulla di paragonabile alla ricerca di spazio libero per l'archiviazione.

Potreste obiettare che l'heap non è un nastro trasportatore: in effetti, considerandolo tale vi ritrovereste a paginare memoria, scrivendola e leggendola su disco, allo scopo di simulare molta più memoria di quanta ne abbiate a disposizione. La tecnica di paginazione della memoria incide notevolmente sulle prestazioni e alla fine, dopo aver creato un certo numero di oggetti, vi ritrovereste con la memoria esaurita. Il trucco è nell'intervento del garbage collector, che mentre lavora compatta tutti gli oggetti nell'heap: questo implica l'effettivo spostamento del "puntatore all'heap" in un punto più vicino all'inizio del nastro, ben lontano da eventuali errori di paginazione. Il garbage collector esegue un riordino e rende possibile un tipo di allocazione di memoria conforme al modello "ad alta velocità", caratterizzato da un "heap infinitamente libero".

Per comprendere il meccanismo di garbage collection di Java, è utile esaminare il funzionamento di questi meccanismi su altri sistemi. Una tecnica semplice, per quanto lenta, di garbage-collection è il cosiddetto *reference counting*, secondo il quale ogni oggetto contiene un contatore di riferimenti che si incrementa ogni volta che un riferimento viene collegato a quell'oggetto; analogamente, ogni volta che un riferimento esce dall'ambito o è impostato a **null**, il contatore viene decrementato. Pertanto la gestione del conteggio dei riferimenti rappresenta un piccolo ma costante onere che si protrae per tutta la durata del programma. Il garbage collector si sposta attraverso l'elenco degli oggetti, e quando ne individua uno con il conteggio dei riferimenti a zero rilascia la memoria che questo occupa; tenete presente, tuttavia, che spesso i meccanismi di reference counting rilasciano un oggetto non appena il conteggio giunge a zero. L'inconveniente principale di questa tecnica è che se alcuni oggetti si autoreferenziano possono dare luogo a un conteggio di riferimenti diverso da zero, pur essendo comunque da eliminare. L'individuazione di questi gruppi autoreferenziati richiede un notevole lavoro supplementare per il garbage collector. Per questi motivi, il conteggio dei riferimenti è comunemente impiegato per illustrare un tipo di garbage collection, ma sembra che non sia utilizzato in nessuna implementazione JVM.

Negli schemi caratterizzati da maggiore velocità la garbage collection non si basa sul conteggio dei riferimenti, bensì sull'assunto che qualsiasi oggetto "non morto" debba essere riconducibile a un riferimento presente nello stack o nell'area di memorizzazione statica. La catena potrebbe attraversare vari livelli di oggetti: in ogni caso, partendo dallo stack o dall'area di memorizzazione statica e attraversando tutti i riferimenti si troveranno tutti gli oggetti attivi. Per ogni riferimento individuato sarà necessario tracciare l'oggetto a cui esso punta e seguire via via tutti i riferimenti presenti a *questo* oggetto, fino ad attraversare l'intera ragnatela di riferimenti creata dal riferimento

presente sullo stack o nell'area di memorizzazione statica. Ogni oggetto attraversato deve essere attivo. Tenete presente che non esiste alcun problema con i gruppi di oggetti auto-referenziati: semplicemente non vengono rilevati e diventano normali oggetti da eliminare.

Con questo approccio la JVM adotta uno schema di garbage collection adattabile, e ciò che avviene agli oggetti attivi identificati dipende dalla variante utilizzata. Una di queste varianti è detta *stop-and-copy* poiché, per ragioni che appariranno presto evidenti, il programma viene per prima cosa interrotto: questo schema non funziona in background. In seguito ogni oggetto attivo viene copiato da un heap a un altro, ignorando gli oggetti inutili. Inoltre, nel momento in cui gli oggetti vengono copiati nel nuovo heap sono "impaccati" dal primo all'ultimo, di fatto compattando il nuovo heap e permettendo così di "srotolare" nuovo spazio disponibile, come descritto in precedenza a proposito del nastro trasportatore.

Chiaramente, quando un oggetto viene spostato da un punto a un altro tutti i riferimenti che puntano a esso devono essere modificati. Il riferimento che va dall'heap o dall'area di memorizzazione statica all'oggetto può essere cambiato all'istante, tuttavia possono esistere altri riferimenti indicanti l'oggetto che saranno incontrati in seguito durante il "tragitto" e corretti via via: consideratela una sorta di tabella di corrispondenza tra indirizzi vecchi e nuovi.

Sono due le considerazioni che rendono poco efficienti questi cosiddetti "collettori di copie". La prima è l'idea di disporre di due heap e spostare avanti e indietro tutta la memoria tra questi, mantenendo così il doppio della memoria effettivamente necessaria. Alcune JVM gestiscono questa situazione assegnando le porzioni di heap via via necessarie e limitandosi a copiare da una porzione all'altra.

La seconda considerazione riguarda il processo di copia in sé. Quando il programma diventa stabile potrebbe generare poca o nessuna "spazzatura": ciononostante un "collettore di copie" continuerà a copiare tutta la memoria da un settore all'altro, originando uno spreco di risorse. Per ovviare a questo inconveniente alcune JVM rilevano quando non viene generata nessuna nuova spazzatura e commutano su un diverso schema: questo è il componente "adattabile", chiamato *mark-and-sweep* e adottato dalle prime versioni della JVM di Sun. Per un utilizzo generico lo schema mark-and-sweep è piuttosto lento, ma quando si sa di produrre poca o nessuna "spazzatura" diventa veloce.

Questo schema segue anch'esso la logica che consiste nel partire dallo stack e dall'area di memorizzazione statica, seguendo tutti i riferimenti agli oggetti per rilevare quelli attivi: a differenza dell'altro, però, ogni volta che trova un



oggetto attivo lo contrassegna con un apposito indicatore, senza raccoglierlo. Soltanto a processo di marcatura completato interviene il cleanup, e gli oggetti inutilizzati vengono eliminati dalla memoria. In ogni caso lo schema mark-and-sweep non esegue processi di copia in modo che, se il collettore decide di compattare un heap frammentato, lo fa rimescolando gli oggetti.

Il termine stop-and-copy sottintende l'idea che questo tipo di garbage collection non viene eseguito in background; durante l'operazione il programma si interrompe. Nella documentazione Sun troverete numerosi riferimenti alla garbage collection come processo di background a bassa priorità, tuttavia non era così nelle versioni iniziali della JVM: il garbage collector di Sun interrompeva il programma quando la memoria disponibile era insufficiente. Anche lo schema mark-and-sweep richiede che il programma venga interrotto.

Come si è visto in precedenza, nella JVM descritta in questo contesto la memoria viene allocata in grandi blocchi: se allocate un oggetto di grandi dimensioni, esso otterrà il proprio blocco. Lo schema stop-and-copy richiede la copia di ogni oggetto attivo dall'heap sorgente a quello nuovo prima di poter liberare il vecchio heap, il che implica la disponibilità di un'ingente quantità di memoria. Mediante l'uso dei blocchi, la garbage collection è normalmente in grado di copiare gli oggetti nei blocchi inattivi non appena li trova; ogni blocco possiede un contatore, detto *generation count*, necessario per verificare se è ancora attivo.

In situazioni normali soltanto i nuovi blocchi creati dall'ultima garbage collection vengono compattati, mentre per tutti gli altri il contatore è azzerato se risulta che sono referenziati in qualche punto. In questo modo viene gestita la situazione standard, che corrisponde alla presenza di molti oggetti di breve durata. Periodicamente Java esegue un cleanup completo, ma i grandi blocchi non vengono ancora copiati: ne viene semplicemente azzerato il contatore, mentre i blocchi contenenti piccoli oggetti sono copiati e compatti. La JVM controlla l'efficienza della garbage collection, e se diventa una perdita di tempo perché tutti gli oggetti hanno vita lunga passa allo schema mark-and-sweep; allo stesso modo la JVM tiene traccia dell'efficienza dello schema mark-and-sweep: se l'heap diventa frammentato, commuta in stop-and-copy. È in questo meccanismo che si spiega il termine "adattabile", parafrasabile in una definizione che riempie la bocca di molti tecnici: *adaptive generational stop-and-copy mark-and-sweep!*

Java consente di implementare alcune tecniche aggiuntive di accelerazione, una delle quali è particolarmente importante e coinvolge l'attività del loader e del cosiddetto compilatore JIT (*Just-In-Time*). Un compilatore JIT converte un programma, in modo parziale o completo, nel codice macchina nativo,



affinché non debba essere interpretato dalla JVM e venga eseguito molto rapidamente. Al momento di caricare una classe, di norma la prima volta che create un suo oggetto, viene recuperato il file **.class** relativo e caricato in memoria il bytecode della classe.

A quel punto un approccio possibile è la compilazione completa del codice da parte del compilatore JIT, operazione che tuttavia presenta due inconvenienti: richiede tempo aggiuntivo che prolunga la durata del programma e incrementa le dimensioni dell'eseguibile (il bytecode è notevolmente più compatto del codice espanso da JIT). Questo potrebbe causare la paginazione in memoria, che rallenterebbe definitivamente il programma. Un metodo alternativo, chiamato *lazy evaluation* (letteralmente, "valutazione pigra"), prevede che il codice non sia compilato dal JIT fino a quando non sia necessario: questo vuol dire che il codice che non venisse mai eseguito potrebbe non essere mai compilato dal compilatore JIT. Le più recenti tecnologie Java HotSpot dei JDK (*Java Development Kit*) adottano un approccio simile, incrementando l'ottimizzazione del codice ogni volta che viene eseguito: quanto più spesso si esegue il codice, tanto più velocemente verrà eseguito.

Inizializzazione dei membri

Java fa del proprio meglio per garantire che le variabili siano inizializzate correttamente prima del loro utilizzo. In caso di variabili locali di un metodo questa garanzia è fornita sotto forma di errori di compilazione; pertanto, scrivendo

```
void f() {  
    int i;  
    i++; // Errore -- i non è inizializzato  
}
```

otterrete un messaggio di errore indicante che **i** potrebbe non essere stato inizializzato. Ovviamente il compilatore avrebbe potuto assegnare a **i** un valore predefinito, ma una variabile locale non inizializzata è quasi sempre un errore di programmazione che un valore predefinito avrebbe coperto; il fatto di costringere il programmatore a fornire un valore di inizializzazione rende più probabile l'individuazione dei bug.

Se il tipo primitivo è un campo di una classe, tuttavia, le cose sono un po' diverse. Come avete visto nel Capitolo 2 ogni campo di tipo primitivo appartenente a una classe è sempre fornito di un valore iniziale. Di seguito è pre-

sentato un programma che verifica questa affermazione e visualizza questi valori iniziali.

```
//: initialization/InitialValues.java
// Mostra i valori iniziali predefiniti.
import static net.mindview.util.Print.*;

public class InitialValues {
    boolean t;
    char c;
    byte b;
    short s;
    int i;
    long l;
    float f;
    double d;
    InitialValues reference;
    void printInitialValues() {
        print("Data type      Initial value");
        print("boolean        " + t);
        print("char          [" + c + "]");
        print("byte          " + b);
        print("short         " + s);
        print("int           " + i);
        print("long          " + l);
        print("float         " + f);
        print("double        " + d);
        print("reference     " + reference);
    }
    public static void main(String[] args) {
        InitialValues iv = new InitialValues();
        iv.printInitialValues();
        /* Potete anche scrivere:
        new InitialValues().printInitialValues();
        */
    }
}
```

```
} /* Output:
Data type      Initial value
boolean        false
char          [ ]
byte          0
short         0
int           0
long          0
float         0.0
double        0.0
reference     null
*///:~
```

Come potete notare, malgrado non siano stati specificati valori i dati vengono automaticamente inizializzati: il valore di **char** è zero, visualizzato come uno spazio. In questo modo, se non altro, non correte il rischio di lavorare con variabili non inizializzate.

Se definite un riferimento a un oggetto all'interno di una classe senza inizializzarlo a un nuovo oggetto, gli verrà assegnato il valore speciale **null**.

Come specificare l'inizializzazione

Come fare per assegnare un valore iniziale a una variabile? Un modo diretto per eseguire tale operazione consiste nella semplice assegnazione del valore al momento della definizione della variabile nella classe: ricordate che questa operazione non è consentita in C++, sebbene i neofiti di questo linguaggio provino sempre a eseguirla. In questo caso la definizione dei campi nella classe **InitialValues** è stata modificata per fornire i valori iniziali:

```
//: initialization/InitialValues2.java
// Fornisce esplicitamente i valori di inizializzazione.

public class InitialValues2 {
    boolean bool = true;
    char ch = 'x';
    byte b = 47;
    short s = 0xff;
    int i = 999;
```



```
long lng = 1;
float f = 3.14f;
double d = 3.14159;
} //:~
```

È possibile inizializzare in questo modo anche gli oggetti non-primitivi. Se **Depth** è una classe, potete generare una variabile e inizializzarla con la tecnica seguente:

```
//: initialization/Measurement.java
class Depth {}

public class Measurement {
    Depth d = new Depth();
    // ...
} //:~
```

Se non fornite a **d** un valore iniziale e provate a utilizzarlo, otterrete un errore di runtime chiamato *eccezione*, che vedrete meglio nel Volume 2, Capitolo 1, dedicato alla gestione degli errori. Potete anche chiamare un metodo che fornisce un valore iniziale:

```
//: initialization/MethodInit.java
public class MethodInit {
    int i = f();
    int f() { return 11; }
} //:~
```

Ovviamente questo metodo può accettare argomenti, che tuttavia non possono essere altri membri della classe che non siano stati ancora inizializzati. Di conseguenza potete scrivere il codice seguente:

```
//: initialization/MethodInit2.java
public class MethodInit2 {
    int i = f();
    int j = g(i);
    int f() { return 11; }
    int g(int n) { return n * 10; }
} //:~
```



Non potete invece scrivere questo codice:

```
//: initialization/MethodInit3.java
public class MethodInit3 {
    //! int j = g(i); // Forward referencing non ammesso
    int i = f();
    int f() { return 11; }
    int g(int n) { return n * 10; }
} //:~
```

Ecco una tipica situazione in cui il compilatore si lamenta, a ragione, del cosiddetto *forward referencing* (riferimento anticipato), che si riferisce non tanto all'ordine di inizializzazione quanto al modo in cui il programma viene compilato.

Come vedete questo approccio all'inizializzazione è semplice e diretto, tuttavia presenta il limite di fornire sempre gli stessi valori di inizializzazione per ogni oggetto di tipo **InitialValues**. Talvolta questo è esattamente ciò che vi aspettate, ma in altre situazioni potreste desiderare una maggiore adattabilità.

Inizializzazione del costruttore

Il costruttore può essere utilizzato per eseguire l'inizializzazione: questo garantisce una maggiore flessibilità di programmazione poiché è possibile chiamare metodi ed eseguire azioni in fase di esecuzione al fine di determinare i valori iniziali. Dovete però ricordare che questo non impedisce l'inizializzazione automatica che Java esegue prima di accedere al costruttore. Quindi, scrivendo per esempio

```
//: initialization/Counter.java
public class Counter {
    int i;
    Counter() { i = 7; }
    // ...
} //:~
```

i verrà prima inizializzato a 0, poi a 7. Questo avviene con tutti i tipi primitivi e i riferimenti a oggetti, inclusi quelli inizializzati in modo esplicito durante

la definizione. Per questo motivo il compilatore non vi obbliga a inizializzare elementi in un punto particolare del costruttore o prima che vengano utilizzati: l'inizializzazione è già garantita.

Ordine di inizializzazione

All'interno di una classe l'ordine di inizializzazione è determinato dalla sequenza con cui sono definite le variabili. Le definizioni delle variabili possono essere distribuite un po' ovunque nelle definizioni dei metodi; in ogni caso le variabili sono inizializzate prima di poter chiamare qualsiasi metodo, incluso il costruttore.

```
//: initialization/OrderOfInitialization.java
// Dimostra l'ordine di inizializzazione.
import static net.mindview.util.Print.*;

// Quando il costruttore viene chiamato per creare
// un oggetto Window, visualizzerà un messaggio:
class Window {
    Window(int marker) { print("Window(" + marker + ")"); }
}

class House {
    Window w1 = new Window(1); // Prima del costruttore
    House() {
        // Dimostra di essere nel costruttore:
        print("House()");
        w3 = new Window(33); // Inizializza nuovamente w3
    }
    Window w2 = new Window(2); // Dopo il costruttore
    void f() { print("f()"); }
    Window w3 = new Window(3); // Al termine
}

public class OrderOfInitialization {
    public static void main(String[] args) {
        House h = new House();
        h.f(); // Mostra il completamento della costruzione
    }
}
```

```
}
} /* Output:
Window(1)
Window(2)
Window(3)
House()
Window(33)
f()
*///:~
```

In **House** le definizioni degli oggetti **Window** sono state volutamente disseminate nel codice per dimostrare che la loro inizializzazione avviene prima dell'accesso al costruttore e di qualsiasi altro evento; inoltre, **w3** è nuovamente inizializzato all'interno del costruttore.

L'output evidenzia che il riferimento **w3** viene inizializzato due volte, appena prima e all'interno della chiamata al costruttore: il primo oggetto viene abbandonato per essere preso in carico più tardi dal garbage collector.

A una prima impressione questa tecnica potrebbe sembrare poco efficiente, tuttavia garantisce una corretta inizializzazione: che cosa accadrebbe, infatti, se un costruttore sovraccarico fosse definito in modo da non inizializzare **w3** e non vi fosse un'inizializzazione "predefinita" per **w3** nella sua definizione?

Inizializzazione di dati static

A prescindere dal numero di oggetti creati esiste un solo punto di memorizzazione per un oggetto **static**. Non è possibile utilizzare la parola chiave **static** con variabili locali, pertanto è applicabile soltanto ai campi. Se un campo è uno **static** di tipo primitivo e non viene inizializzato, otterrà il valore di inizializzazione standard previsto dal suo tipo; se è un riferimento a un oggetto il valore di inizializzazione predefinito sarà **null**.

Qualora vogliate fornire un'inizializzazione al momento della definizione, essa dovrà essere identica a quella degli oggetti non statici.

Per vedere quando viene inizializzata la memorizzazione **static** prendete in esame l'esempio seguente:

```
//: initialization/StaticInitialization.java
// Vengono specificati i valori iniziali in una definizione di
// classe.
import static net.mindview.util.Print.*;
```



```
class Bowl {  
    Bowl(int marker) {  
        print("Bowl(" + marker + ")");  
    }  
    void f1(int marker) {  
        print("f1(" + marker + ")");  
    }  
}  
  
class Table {  
    static Bowl bowl1 = new Bowl(1);  
    Table() {  
        print("Table()");  
        bowl2.f1(1);  
    }  
    void f2(int marker) {  
        print("f2(" + marker + ")");  
    }  
    static Bowl bowl2 = new Bowl(2);  
}  
  
class Cupboard {  
    Bowl bowl3 = new Bowl(3);  
    static Bowl bowl4 = new Bowl(4);  
    Cupboard() {  
        print("Cupboard()");  
        bowl4.f1(2);  
    }  
    void f3(int marker) {  
        print("f3(" + marker + ")");  
    }  
    static Bowl bowl5 = new Bowl(5);  
}  
  
public class StaticInitialization {  
    public static void main(String[] args) {
```

5 • Inizializzazione e cleanup

```
    print("Creating new Cupboard() in main");  
    new Cupboard();  
    print("Creating new Cupboard() in main");  
    new Cupboard();  
    table.f2(1);  
    cupboard.f3(1);  
}  
static Table table = new Table();  
static Cupboard cupboard = new Cupboard();  
} /* Output:  
Bowl(1)  
Bowl(2)  
Table()  
f1(1)  
Bowl(4)  
Bowl(5)  
Bowl(3)  
Cupboard()  
f1(2)  
Creating new Cupboard() in main  
Bowl(3)  
Cupboard()  
f1(2)  
Creating new Cupboard() in main  
Bowl(3)  
Cupboard()  
f1(2)  
f2(1)  
f3(1)  
*///:~
```

Bowl consente di visualizzare la creazione della classe, mentre in **Table** e **Cupboard** i membri **static** di **Bowl** sono distribuiti nelle rispettive definizioni di classe; notate che **Cupboard** crea un oggetto non **static Bowl**, di nome **bowl3**, prima delle definizioni **static**.

Dall'output potete rilevare che l'inizializzazione **static** viene eseguita solo se necessaria. Se non create un oggetto **Table** e non fate mai riferimento a **Ta-**

`ble.bowl1` o `Table.bowl2`, gli static `Bowl` chiamati `bowl1` e `bowl2` non verranno mai creati. Essi vengono inizializzati solo al momento della creazione del primo oggetto `Table` o quando si verifica il primo accesso static; fatto questo, gli oggetti static non vengono inizializzati di nuovo.

Come dimostrato dall'output, l'ordine d'inizializzazione è il seguente: innanzitutto i membri static che non siano già stati inizializzati da una precedente creazione dell'oggetto, poi gli oggetti non static.

Al fine di eseguire il metodo static `main()` la classe `StaticInitialization` deve essere caricata e i suoi campi static `table` e `cupboard` successivamente inizializzati, il che implica il caricamento di queste classi; dal momento che queste ultime contengono oggetti static di tipo `Bowl`, anche `Bowl` viene caricato. Ne deriva che tutte le classi presenti in questo programma vengono caricate prima dell'esecuzione di `main()`. È evidente che quello esaminato non è un caso normale, dal momento che di solito non farete in modo che tutto sia collegato mediante static come in questo esempio.

Per ricapitolare il processo di creazione di un oggetto, prendete in esame una classe chiamata `Dog`.

1. Anche se non richiede esplicitamente la parola chiave static il costruttore è effettivamente un metodo static. Di conseguenza, la prima volta che viene creato un oggetto di tipo `Dog` o che si accede a un campo o a un metodo static della classe `Dog` l'interprete Java deve individuare `Dog.class`, cercandolo nel percorso di sistema (`classpath`).
2. Al caricamento di `Dog.class`, come vedrete in seguito, mediante la creazione di un oggetto `Class` vengono eseguiti tutti i relativi inizializzatori static. Quindi l'inizializzazione static avviene soltanto una volta, al primo caricamento dell'oggetto `Class`.
3. Quando create un nuovo oggetto con `new Dog()` il processo di costruzione dell'oggetto si incarica innanzitutto di allocare la memoria sufficiente per memorizzare un oggetto `Dog` nell'heap.
4. Questa memoria viene azzerata, con impostazione automatica di tutti i tipi primitivi dell'oggetto `Dog` ai loro valori predefiniti: zero per i numeri e l'equivalente per `boolean` e `char`; i riferimenti sono impostati a `null`.
5. Vengono eseguite tutte le inizializzazioni che avvengono al momento della definizione.
6. Infine vengono eseguiti i costruttori. Come vedrete nel Capitolo 7, dedicato al riutilizzo delle classi, questo potrebbe comportare una notevole mole di attività, soprattutto se è chiamata in causa l'ereditarietà.

Inizializzazione static esplicita

Java consente di raggruppare in una classe altre inizializzazioni static all'interno di una speciale clausola static, chiamata talvolta blocco static:

```
//: initialization/Spoon.java
public class Spoon {
    static int i;
    static {
        i = 47;
    }
} ///:~
```

Sembrerebbe trattarsi di un metodo, ma in realtà è soltanto la parola chiave static seguita da un blocco di codice. Come altre inizializzazioni di tipo static questo codice viene eseguito una sola volta: la prima volta che create un oggetto o che accedete a un membro static di quella classe, anche se non create mai un oggetto per essa.

```
//: initialization/ExplicitStatic.java
// Inizializzazione static esplicita, con la clausola "static"
import static net.mindview.util.Print.*;

class Cup {
    Cup(int marker) {
        print("Cup(" + marker + ")");
    }
    void f(int marker) {
        print("f(" + marker + ")");
    }
}

class Cups {
    static Cup cup1;
    static Cup cup2;
    static {
        cup1 = new Cup(1);
        cup2 = new Cup(2);
    }
}
```

```

}
Cups() {
    print("Cups()");
}
}

public class ExplicitStatic {
    public static void main(String[] args) {
        print("Inside main()");
        Cups.cup1.f(99); // (1)
    }
    // static Cups cups1 = new Cups(); // (2)
    // static Cups cups2 = new Cups(); // (2)
} /* Output:
Inside main()
Cup(1)
Cup(2)
f(99)
*///:~

```

Gli inizializzatori **static** per **Cups** vengono eseguiti quando l'accesso all'oggetto **static cup1** avviene nella riga contrassegnata da **(1)** o se la riga **(1)** è commentata e dalle righe **(2)** vengono eliminati i commenti. Come potete notare dall'output, se entrambe le sezioni **(1)** e **(2)** sono commentate l'inizializzazione **static** per **Cups** non si verifica mai.

Tenete presente, inoltre, che non ha importanza se una o entrambe le righe contrassegnate da **(2)** sono commentate: l'inizializzazione **static** avviene soltanto una volta.

Esercizio 13 (1) Verificate le affermazioni del paragrafo precedente.

Esercizio 14 (1) Create una classe con un campo **static** di tipo **String** che viene inizializzato al momento della definizione, e un altro campo inizializzato da un blocco **static**. Aggiungete un metodo **static** per visualizzare entrambi i campi e mostrare che essi vengono inizializzati prima dell'utilizzo.

Inizializzazione di istanze non static

Per inizializzare le variabili non statiche di ciascun oggetto, Java mette a disposizione una sintassi simile alla precedente, chiamata *inizializzazione di istanza*. Osservate l'esempio seguente:

```

//: initialization/Mugs.java
// Inizializzazione di istanza in Java.
import static net.mindview.util.Print.*;

class Mug {
    Mug(int marker) {
        print("Mug(" + marker + ")");
    }
    void f(int marker) {
        print("f(" + marker + ")");
    }
}

public class Mugs {
    Mug mug1;
    Mug mug2;
    {
        mug1 = new Mug(1);
        mug2 = new Mug(2);
        print("mug1 & mug2 initialized");
    }
    Mugs() {
        print("Mugs()");
    }
    Mugs(int i) {
        print("Mugs(int)");
    }
    public static void main(String[] args) {
        print("Inside main()");
        new Mugs();
        print("new Mugs() completed");
    }
}

```



```
new Mugs(1);
print("new Mugs(1) completed");
}
/* Output:
Inside main()
Mug(1)
Mug(2)
mug1 & mug2 initialized
Mugs()
new Mugs() completed
Mug(1)
Mug(2)
mug1 & mug2 initialized
Mugs(int)
new Mugs(1) completed
*//*:~
```

Noteate che la clausola di inizializzazione di istanza

```
{
    mug1 = new Mug(1);
    mug2 = new Mug(2);
    print("mug1 & mug2 initialized");
}
```

ricorda quasi esattamente la clausola di inizializzazione statica, da cui differisce per l'assenza della parola chiave **static**. Questa sintassi è indispensabile per supportare l'inizializzazione delle *classi interne anonime*, che vedrete nel Capitolo 10, dedicato alle classi interne; inoltre permette di garantire l'esecuzione di determinate operazioni, a prescindere dal costruttore esplicito che viene chiamato. Dall'output potete vedere che la clausola di inizializzazione di istanza viene eseguita prima della chiamata a uno dei costruttori.

Esercizio 15 (1) Create una classe con una **String** che viene inizializzata mediante l'inizializzazione di istanza.



Inizializzazione di array

Un array non è altro che una sequenza di oggetti o tipi primitivi, tutti dello stesso tipo, assemblati insieme e contrassegnati da uno stesso nome identificativo; gli array vengono definiti e utilizzati con le parentesi quadre, il cosiddetto operatore di indicizzazione **[]**. Per definire un riferimento a un array, scrivete il nome del tipo facendolo seguire da una coppia di parentesi quadre vuote:

```
int[] a1;
```

Potete scrivere le parentesi quadre anche dopo l'identificativo e otterrete esattamente lo stesso significato:

```
int a1[];
```

Questo comportamento è conforme alle attese dei programmati C/C++. Tuttavia lo stile precedente ha una sintassi probabilmente più ragionevole, che definisce il tipo come "un array di **int**": per questo motivo, sarà lo stile adottato nel manuale.

Il compilatore non consente di specificare la dimensione dell'array e ciò riporta ancora una volta all'argomento "riferimenti". A questo punto tutto ciò che avete a disposizione è un riferimento a un array: avete assegnato spazio sufficiente per il riferimento ma non ancora l'oggetto array vero e proprio. Per creare lo spazio di archiviazione per l'array dovete scrivere un'espressione di inizializzazione; nel caso degli array, l'inizializzazione può avvenire ovunque nel codice, ma è anche possibile utilizzare un tipo di espressione particolare che deve essere scritta nel punto in cui viene generato l'array: questa espressione speciale non è altro che un elenco di valori da assegnare, posto tra parentesi graffe. L'allocazione della memoria necessaria, equivalente all'utilizzo dell'operatore **new**, in questo caso è eseguita dal compilatore. Per esempio:

```
int[] a1 = { 1, 2, 3, 4, 5 };
```

Dunque, per quale ragione dovreste definire un riferimento a un array senza un array?

```
int[] a2;
```

Tenete presente che in Java è possibile assegnare un array a un altro, scrivendo:

```
a2 = a1;
```

Tuttavia ciò che state effettivamente eseguendo è la copia di un riferimento, come dimostra il codice seguente:

```
//: initialization/ArraysOfPrimitives.java
import static net.mindview.util.Print.*;

public class ArraysOfPrimitives {
    public static void main(String[] args) {
        int[] a1 = { 1, 2, 3, 4, 5 };
        int[] a2;
        a2 = a1;
        for(int i = 0; i < a2.length; i++)
            a2[i] = a2[i] + 1;
        for(int i = 0; i < a1.length; i++)
            print("a1[" + i + "] = " + a1[i]);
    }
} /* Output:
a1[0] = 2
a1[1] = 3
a1[2] = 4
a1[3] = 5
a1[4] = 6
*//*:~
```

Potete osservare che all'array **a1** viene passato un valore iniziale mentre all'array **a2** no; **a2** viene assegnato in seguito, in questo caso a un altro array. Dal momento che **a2** e **a1** vengono poi associati in alias allo stesso array, le modifiche eseguite in **a2** sono visibili anche in **a1**.

Che si tratti di array di oggetti o di tipi primitivi, tutti hanno un membro intrinseco che potete interrogare, ma non modificare, per sapere di quanti elementi è composto l'array: questo membro è **length**. Poiché in Java, come in C e C++, gli array iniziano a contare i propri elementi da zero l'indice massimo consentito è **length -1**. Se eccedete i limiti dell'array, C e C++ accetteranno il fatto senza eccepire alcunché permettendovi di rovistare ovunque nella memoria: un comportamento che è all'origine di molti famigerati bug; al contrario, Java vi cautela da questa eventualità producendo un errore

di runtime, ossia un'eccezione, ogni qualvolta tenterete di uscire dai limiti dell'array.⁵

Se al momento di scrivere il programma non sapete di quanti elementi avrete bisogno nel vostro array, potete semplicemente ricorrere a **new** per crearli. In questo caso **new** funziona correttamente anche se si sta creando un array di tipi primitivi; naturalmente **new** non creerà un tipo primitivo vero e proprio.

```
//: initialization/ArrayNew.java
// Creazione di array con new.
import java.util.*;
import static net.mindview.util.Print.*;

public class ArrayNew {
    public static void main(String[] args) {
        int[] a;
        Random rand = new Random(47);
        a = new int[rand.nextInt(20)];
        print("length of a = " + a.length);
        print(Arrays.toString(a));
    }
} /* Output:
length of a = 18
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
*//*:~
```

Le dimensioni dell'array vengono scelte casualmente tramite il metodo **Random.nextInt()**, che produce un valore compreso tra zero e quello del suo argomento. La sua casualità dimostra in modo evidente che la creazione dell'array avviene effettivamente durante l'esecuzione; inoltre l'output di questo programma mo-

5. Controllare che il programmatore non ecceda i limiti dell'array ha un costo gestionale significativo e più oneroso in termini di tempo che in C++; questo controllo inoltre non è disattivabile: questo significa che, se si verifica in un punto critico, l'accesso agli array potrebbe diventare un fattore di inefficienza nei vostri programmi. Tenuto conto della sicurezza di Internet e della produttività dei programmati, i progettisti Java hanno ritenuto che fosse un compromesso accettabile. Malgrado possiate essere tentati di scrivere codice che riteneate possa rendere più efficiente l'accesso agli array, sappiate che è una vera e propria perdita di tempo poiché le ottimizzazioni automatiche che Java esegue in fase di compilazione e di esecuzione già velocizzano questi accessi.

stra che gli elementi dell'array di tipi primitivi vengono inizializzati automaticamente con valori "vuoti": per tipi numerici e **char** a zero, per i **boolean** a **false**. Il metodo **Arrays.toString()**, disponibile nella libreria standard **java.util**, genera la versione visualizzabile di un array monodimensionale.

Ovviamente in questo caso l'array potrebbe anche essere stato definito e inizializzato con una sola istruzione:

```
int[] a = new int[rand.nextInt(20)];
```

Questa è la tecnica raccomandata per eseguire tali operazioni, quando ciò sia possibile.

In realtà, quando create un array non primitivo state creando un array di riferimenti. Considerate il tipo wrapper **Integer**, che è una classe e non un tipo primitivo.

```
//: initialization/ArrayClassObj.java
// Creazione di un array di oggetti non primitivi.
import java.util.*;
import static net.mindview.util.Print.*;

public class ArrayClassObj {
    public static void main(String[] args) {
        Random rand = new Random(47);
        Integer[] a = new Integer[rand.nextInt(20)];
        print("length of a = " + a.length);
        for(int i = 0; i < a.length; i++)
            a[i] = rand.nextInt(500); // Autoboxing
        print(Arrays.toString(a));
    }
} /* Output: (esempio)
length of a = 18
[55, 193, 361, 461, 429, 368, 200, 22, 207, 288, 128, 51, 89,
309, 278, 498, 361, 20]
*///:~
```

In questo caso, anche dopo avere chiamato **new** per creare l'array

```
Integer[] a = new Integer[rand.nextInt(20)];
```

si ottiene soltanto un array di riferimenti, e l'inizializzazione non è completa finché il riferimento in sé non viene inizializzato tramite la creazione di un nuovo oggetto **Integer**; in questo esempio, è utilizzato l'autoboxing:

```
a[i] = rand.nextInt(500);
```

Tuttavia, se dimenticate di creare l'oggetto non appena proverete a utilizzare una posizione vuota dell'array otterrete un'eccezione in fase di esecuzione.

È anche possibile inizializzare gli array includendo un elenco tra parentesi graffe, mediante due diverse modalità:

```
//: initialization/ArrayInit.java
// Inizializzazione degli array.
import java.util.*;
```

```
public class ArrayInit {
    public static void main(String[] args) {
        Integer[] a = {
            new Integer(1),
            new Integer(2),
            3, // Autoboxing
        };
        Integer[] b = new Integer[]{
            new Integer(1),
            new Integer(2),
            3, // Autoboxing
        };
        System.out.println(Arrays.toString(a));
        System.out.println(Arrays.toString(b));
    }
} /* Output:
[1, 2, 3]
[1, 2, 3]
*///:~
```

In entrambi i casi la virgola finale nell'elenco di valori di inizializzazione è faticativa; questa caratteristica è stata prevista per semplificare la manutenzione di lunghi elenchi.

Benché la prima forma sia utile, è più limitata poiché utilizzabile soltanto nel punto in cui viene definito l'array. Potete utilizzare la seconda e la terza forma ovunque, anche nell'ambito di una chiamata di metodo. Per esempio potreste creare un insieme di oggetti **String** per passare al **main()** di un altro metodo, allo scopo di fornire argomenti da riga di comando alternativi a quel **main()**:

```
//: initialization/DynamicArray.java
// Inizializzazione degli array.

public class DynamicArray {
    public static void main(String[] args) {
        Other.main(new String[]{"fiddle", "de", "dum"});
    }
}

class Other {
    public static void main(String[] args) {
        for(String s : args)
            System.out.print(s + " ");
    }
    /* Output:
       fiddle de dum
    */
}
```

L'array creato come argomento del metodo **Other.main()** viene generato nel punto in cui è chiamato il metodo, in modo che possiate fornire anche altri argomenti al momento della chiamata.

Esercizio 16 (1) Create un array di oggetti **String**, assegnate una stringa a ciascun elemento e visualizzate l'array ricorrendo a un ciclo **for**.

Esercizio 17 (2) Create una classe tramite un costruttore che accetta un argomento di tipo **String**, e all'interno del costruttore visualizzate l'argomento. Create un insieme di riferimenti a oggetti di questa classe, senza tuttavia creare oggetti da assegnare nell'array. Quando eseguite il programma verificate se i messaggi di inizializzazione delle chiamate al costruttore vengono visualizzati.

Esercizio 18 (1) Completate l'esercizio precedente creando oggetti da collegare all'array di riferimenti.

Elenchi di argomenti variabili

La seconda forma di inizializzazione degli array è caratterizzata da una sintassi utile per creare e chiamare metodi in grado di produrre un effetto analogo agli *elenchi di argomenti variabili* caratteristici del linguaggio C, conosciuti anche come *vararg*. Questi possono includere quantità sconosciute di argomenti, nonché tipi sconosciuti. Poiché tutte le classi sono ereditate dall'oggetto radice **Object**, che vedrete meglio nel prosieguo del manuale, potete creare un metodo che accetta un array di **Object** e chiamarlo nel modo seguente:

```
//: initialization/VarArgs.java
// Uso della sintassi degli array per creare elenchi di
// argomenti variabili.

class A {}

public class VarArgs {
    static void printArray(Object[] args) {
        for(Object obj : args)
            System.out.print(obj + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        printArray(new Object[]{
            new Integer(47), new Float(3.14), new Double(11.11)
        });
        printArray(new Object[]{"one", "two", "three"});
        printArray(new Object[]{new A(), new A(), new A()});
    }
} /* Output: (Sample)
   47 3.14 11.11
   one two three
   A@1a46e30 A@3e25a5 A@19821f
*/
```

Potete vedere che **print()** accetta un array di **Object**, quindi lo elabora tramite la sintassi **foreach** e visualizza ogni elemento. Le classi della libreria standard Java

producono un output sensato, tuttavia gli oggetti delle classi create in questo codice visualizzano il nome della classe seguito dal simbolo @ e da cifre esadecimali. Pertanto il comportamento predefinito, che si verifica se non prevedete un metodo **toString()** per la classe (operazione descritta in seguito), consiste nel visualizzare il nome della classe e l'indirizzo dell'oggetto.

Potreste trovare codice precedente a Java SE5 che impiega questo meccanismo per produrre elenchi di argomenti variabili. In Java SE5, però, è stata finalmente integrata questa caratteristica a lungo richiesta; ora potete utilizzare i puntini di sospensione per definire un elenco di argomenti variabile, come potete vedere in **printArray()**:

```
//: initialization/NewVarArgs.java
/* Uso della sintassi degli array per creare elenchi di
   argomenti variabili.*/

public class NewVarArgs {
    static void printArray(Object... args) {
        for(Object obj : args)
            System.out.print(obj + " ");
        System.out.println();
    }

    public static void main(String[] args) {
        // Puo' accettare elementi singoli:
        printArray(new Integer(47), new Float(3.14),
                  new Double(11.11));
        printArray(47, 3.14F, 11.11);
        printArray("one", "two", "three");
        printArray(new A(), new A(), new A());
        // Oppure un array:
        printArray((Object[])new Integer[]{ 1, 2, 3, 4 });
        printArray(); // L'elenco vuoto e' OK
    }

} /* Output: (75% match)
47 3.14 11.11
47 3.14 11.11
one, two, three ....
A@1bab50a A@c3c749 A@150bd4d
*///:~
```

1 2 3 4

*///:~

Con i *vararg* non dovete più scrivere esplicitamente la sintassi dell'array: il compilatore infatti lo completerà quando specificherete argomenti variabili. Ottenete ancora un array, poiché **print()** è in grado di usare la sintassi **foreach** per iterarlo, tuttavia l'operazione è qualcosa di più di una semplice conversione automatica da un elenco di elementi a un array. Notate la penultima riga del programma, in cui un array di **Integer**, creato mediante l'autoboxing, viene convertito in un array di **Object** per rimuovere un avvertimento del compilatore, e poi è passato al metodo **printArray()**. Chiaramente il compilatore vede che questo è già un array e non esegue la conversione. Se avete un gruppo di elementi potete passarli sotto forma di elenco; se avete già un array anch'esso sarà accettato come elenco di argomenti variabile.

L'ultima riga del programma mostra che è possibile anche non passare alcun argomento a un elenco vararg, un accorgimento che si rivela utile qualora vogliate gestire argomenti facoltativi:

```
//: initialization/OptionalTrailingArguments.java

public class OptionalTrailingArguments {
    static void f(int required, String... trailing) {
        System.out.print("required: " + required + " ");
        for(String s : trailing)
            System.out.print(s + " ");
        System.out.println();
    }

    public static void main(String[] args) {
        f(1, "one");
        f(2, "two", "three");
        f(0);
    }

} /* Output:
required: 1 one
required: 2 two three
required: 0
*///:~
```



L'esempio precedente mostra anche come utilizzare vararg con un tipo specifico diverso da **Object**. In questo caso tutti i vararg devono essere composti di oggetti **String**: è comunque possibile utilizzare qualsiasi tipo di argomento, inclusi i tipi primitivi. L'esempio che segue mostra anche che l'elenco vararg si trasforma in un array, e che se tale elenco è vuoto si tratta di un array di dimensione zero:

```
//: initialization/VarargType.java

public class VarargType {
    static void f(Character... args) {
        System.out.print(args.getClass());
        System.out.println(" length " + args.length);
    }
    static void g(int... args) {
        System.out.print(args.getClass());
        System.out.println(" length " + args.length);
    }
    public static void main(String[] args) {
        f('a');
        f();
        g(1);
        g();
        System.out.println("int[]: " + new int[0].getClass());
    }
} /* Output:
class [Ljava.lang.Character; length 1
class [Ljava.lang.Character; length 0
class [I length 1
class [I length 0
int[]: class [I
*//:~
```

Il metodo **getClass()**, che fa parte di **Object** e sarà trattato nel Volume 2, Capitolo 2, dedicato alle informazioni sui tipi, restituisce la classe di un oggetto: visualizzando ciò che viene restituito da questa classe vedrete una stringa codificata che rappresenta il tipo di classe. La parentesi quadra aperta iniziale “[” indica che si tratta di un array del tipo specificato subito dopo. Il carattere “I” indica il tipo primitivo **int**; per un’ulteriore verifica viene creato un array di **int**



nell’ultima riga e ne viene visualizzato il tipo. Questo permette di constatare che l’utilizzo di vararg non dipende dalla funzionalità autoboxing ma che in effetti utilizza i tipi primitivi.

I vararg, in ogni caso, funzionano in armonia con l’autoboxing.

```
//: initialization/AutoboxingVarargs.java

public class AutoboxingVarargs {
    public static void f(Integer... args) {
        for(Integer i : args)
            System.out.print(i + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        f(new Integer(1), new Integer(2));
        f(4, 5, 6, 7, 8, 9);
        f(10, new Integer(11), 12);
    }
} /* Output:
1 2
4 5 6 7 8 9
10 11 12
*//:~
```

Noteate che è permesso mescolare i diversi tipi in un solo elenco di argomenti; in modo selettivo l’autoboxing promuove gli argomenti **int** a **Integer**.

I vararg complicano il processo di overloading, per quanto a prima vista sembri sufficientemente sicuro.

```
//: initialization/OverloadingVarargs.java

public class OverloadingVarargs {
    static void f(Character... args) {
        System.out.print("first");
        for(Character c : args)
            System.out.print(" " + c);
        System.out.println();
    }
}
```



```

}
static void f(Integer... args) {
    System.out.print("second");
    for(Integer i : args)
        System.out.print(" " + i);
    System.out.println();
}
static void f(Long... args) {
    System.out.println("third");
}
public static void main(String[] args) {
    f('a', 'b', 'c');
    f(1);
    f(2, 1);
    f(0);
    f(0L);
    //! f(); // Non puo' compilare perche' ambiguo
}
/* Output:
first a b c
second 1
second 2 1
second 0
third
*///:~

```

In tutti i casi, il compilatore si serve dell'autoboxing per attuare la corrispondenza al metodo sovraccarico e chiamare il metodo che corrisponde in modo più specifico. Tuttavia quando chiamate il metodo `f()` senza argomenti, il compilatore non ha alcun modo per determinare il metodo da chiamare; malgrado questo errore sia comprensibile, probabilmente sorprenderà il programmatore client. Potreste cercare di risolvere il problema aggiungendo un argomento non vararg a uno dei metodi.

```
//: initialization/OverloadingVarargs2.java
// {CompileTimeError} (Non puo' compilare)
```



```

public class OverloadingVarargs2 {
    static void f(float i, Character... args) {
        System.out.println("first");
    }
    static void f(Character... args) {
        System.out.print("second");
    }
    public static void main(String[] args) {
        f(1, 'a');
        f('a', 'b');
    }
} //:~

```

Il tag di commento `{CompileTimeError}` esclude questo file dalla funzionalità di build Ant prevista per questo manuale. Tuttavia, compilando manualmente il codice otterrete il messaggio di errore seguente:

```
reference to f is ambiguous, both method f(float,java.lang.Character...)
in OverloadingVarargs2 and method f(java.lang.Character...) in Over-
loadingVarargs2match]
```

Se invece passate a entrambi i metodi un argomento non vararg, il codice funzionerà regolarmente.

```
//: initialization/OverloadingVarargs3.java

public class OverloadingVarargs3 {
    static void f(float i, Character... args) {
        System.out.println("first");
    }
    static void f(char c, Character... args) {
        System.out.println("second");
    }
    public static void main(String[] args) {
        f(1, 'a');
        f('a', 'b');
    }
} /* Output:
```

```
first
second
*///:~
```

In generale dovreste utilizzare un elenco variabile di argomenti soltanto su una versione di un metodo sovraccarico, oppure prendere in considerazione di non servirne affatto.

Esercizio 19 (2) Scrivete un metodo che accetta un array **String** vararg. Verificate di potere passare a questo metodo un elenco di **String** separato da virgolette oppure una **String[]**.

Esercizio 20 (1) Create un metodo **main()** che utilizza i vararg invece della normale sintassi di **main()**. Visualizzate tutti gli elementi contenuti nell'array **args** risultante; provatelo con numeri diversi di argomenti da riga di comando.

Tipi enumerativi

Un'integrazione apparentemente trascurabile di Java SE5 è la parola chiave **enum**, che tuttavia semplificherà il vostro lavoro di programmatore quando avrete bisogno di raggruppare e utilizzare un insieme di *tipi enumerativi*. In passato avreste dovuto creare un insieme di valori interi costanti, che però non si adattano con naturalezza al vostro insieme e sono più rischiosi e difficili da utilizzare. I tipi enumerativi sono una necessità talmente ricorrente che C, C++ e molti altri linguaggi li mettono a disposizione da sempre. Prima di Java SE5 i programmatori Java erano costretti a prestare molta attenzione nel riprodurre correttamente l'effetto di **enum**. Ora anche Java ha la propria funzionalità **enum**, molto più completa di quella disponibile in C e C++. Considerate questo semplice esempio:

```
//: initialization/Spiciness.java
public enum Spiciness {
    NOT, MILD, MEDIUM, HOT, FLAMING
} //://:~
```

Questo codice crea un tipo enumerativo chiamato **Spiciness** con cinque valori specifici che corrispondono ad altrettante gradazioni del sapore piccante. Poiché le istanze di tipi enumerativi sono costanti, per convenzione vengono indicate in lettere maiuscole; se un nome è composto di più parole dovrete separarle da caratteri di sottolineatura.

Per utilizzare una **enum** basta creare un riferimento di questo tipo e assegnarlo a un'istanza.

```
//: initialization/SimpleEnumUse.java
public class SimpleEnumUse {
    public static void main(String[] args) {
        Spiciness howHot = Spiciness.MEDIUM;
        System.out.println(howHot);
    }
} /* Output:
MEDIUM
*///:~
```

Quando create una **enum** il compilatore aggiunge automaticamente alcune caratteristiche utili; per esempio, genera un metodo **toString()** per consentire di visualizzare facilmente il nome di un'istanza **enum**: questo è il motivo per cui l'istruzione **print** dell'esempio precedente ha prodotto l'output. Il compilatore crea anche un metodo **ordinal()** che indica l'ordine di dichiarazione di una determinata costante **enum**, e un metodo **values()** statico che produce un array di valori delle costanti **enum**, nell'ordine in cui sono state dichiarate.

```
//: initialization/EnumOrder.java
public class EnumOrder {
    public static void main(String[] args) {
        for(Spiciness s : Spiciness.values())
            System.out.println(s + ", ordinal " + s.ordinal());
    }
} /* Output:
NOT, ordinal 0
MILD, ordinal 1
MEDIUM, ordinal 2
HOT, ordinal 3
FLAMING, ordinal 4
*///:~
```

Sebbene sembri corrispondere a un nuovo tipo di dato, la parola chiave **enum** si limita a innescare particolari comportamenti del compilatore durante la generazione di una classe per l'**enum**, in modo che per molti aspetti possiate gestire una

enum come se si trattasse di una classe qualsiasi. In effetti le **enum** sono classi e sono dotate di metodi propri.

Una caratteristica particolarmente interessante è il modo in cui una **enum** può essere utilizzata all'interno di un'istruzione **switch**.

```
//: initialization/Burrito.java

public class Burrito {
    Spiciness degree;
    public Burrito(Spiciness degree) { this.degree = degree; }
    public void describe() {
        System.out.print("This burrito is ");
        switch(degree) {
            case NOT:    System.out.println("not spicy alt all.");
                          break;
            case MILD:
            case MEDIUM: System.out.println("a little hot.");
                           break;
            case HOT:
            case FLAMING:
            default:     System.out.println("maybe too hot.");
        }
    }
    public static void main(String[] args) {
        Burrito
        plain = new Burrito(Spiciness.NOT),
        greenChile = new Burrito(Spiciness.MEDIUM),
        jalapeno = new Burrito(Spiciness.HOT);
        plain.describe();
        greenChile.describe();
        jalapeno.describe();
    }
} /* Output:
This burrito is not spicy alt all.
This burrito is a little hot.
This burrito is maybe too hot.
*///:~
```

Poiché l'istruzione **switch** è progettata per operare una selezione da un insieme di possibilità limitato, essa rappresenta la corrispondenza ideale per una **enum**. Osservate come i nomi **enum** possano chiarire meglio ciò che il programma deve eseguire.

Di norma potete utilizzare una **enum** come se fosse un altro modo per creare un tipo di dato e semplicemente metterne il risultato all'opera. Questo è il punto, in effetti: non dovete curarvi troppo della loro presenza. Prima dell'introduzione delle **enum** in Java SE5 era necessario impegnarsi a fondo per ottenere un tipo enumerativo equivalente, il cui utilizzo fosse abbastanza sicuro.

Per il momento questo è sufficiente per farvi capire e utilizzare le **enum** di base; l'argomento verrà comunque approfondito nel Volume 2, Capitolo 7, dedicato specificamente ai tipi enumerativi.

Esercizio 21 (1) Create una **enum** dei sei tagli di banconote caratterizzati dal valore più basso, quindi eseguite un'iterazione in **values()** e visualizzate ogni valore e il suo **ordinal()**.

Esercizio 22 (2) Scrivete un'istruzione **switch** per l'**enum** dell'esempio precedente; per ogni caso visualizzate una descrizione di quella particolare valuta.

Riepilogo

Il fatto che il costruttore sia un meccanismo così (apparentemente) elaborato è sintomatico di quanto Java consideri critica la fase di inizializzazione. Quando Bjarne Stroustrup, l'inventore del C++, progettava questo linguaggio, una delle prime osservazioni che fece sulla produttività in C era che l'errata inizializzazione delle variabili genera una parte significativa dei problemi di programmazione. È difficile identificare questo tipo di bug, e analoghe considerazioni sono valide anche in caso di cleanup improprio della memoria. Dal momento che i costruttori assicurano la correttezza dell'inizializzazione e del cleanup, in virtù del fatto che il compilatore non permette di creare un oggetto senza le opportune chiamate al suo costruttore, potete aspettarvi un livello assoluto di controllo e sicurezza.

In C++ la distruzione è un'operazione importante poiché gli oggetti creati con **new** devono essere distrutti in modo esplicito. In Java, invece, il garbage collector rilascia automaticamente la memoria per tutti gli oggetti, cosicché di solito non occorre utilizzare un metodo di cleanup specifico: quando però è necessario, dovete crearlo voi stessi. Nei casi in cui non sia richiesto un comportamento analogo a quello del distruttore, il garbage collector di

Java semplifica la programmazione e incrementa la sicurezza, essenziale nella gestione della memoria. Alcuni meccanismi di garbage collection sono in grado di "ripulire" anche risorse di altro tipo, quali la grafica e gli handle di file. Il garbage collector, tuttavia, aggiunge un onere elaborativo durante l'esecuzione, difficile da quantificare in considerazione della lentezza "storica" degli interpreti Java. Benché nel corso degli anni Java abbia incrementato le proprie prestazioni in modo notevole, per alcuni tipi di problemi di programmazione il fattore velocità ha sempre influito sul linguaggio.

Tenuto conto del fatto che la costruzione di qualunque oggetto viene garantita, sui costruttori si potrebbe dire molto di più di quanto è stato presentato in questo capitolo. In particolare, quando create nuove classi utilizzando la composizione o l'ereditarietà, la garanzia di costruzione permane, rendendo tuttavia necessario qualche aggiustamento sintattico. Nei prossimi capitoli affronterete la composizione, l'ereditarietà e il modo in cui queste funzionalità influiscono sui costruttori.

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

Capitolo 6

Controllo di accesso



Il concetto di controllo di accesso, detto anche occultamento dell'implementazione o *implementation hiding*, è assimilato all'idea che "non tutto riesce al primo colpo".

Tutti i migliori scrittori, compresi coloro che scrivono software, sanno che un lavoro non è ben riuscito fino a quando non è stato riscritto, spesso molte volte. Se lasciate un frammento di codice nel cassetto per qualche tempo e poi lo rileggete, potreste scoprire che le stesse operazioni possono essere eseguite in modo migliore. Questa è una delle motivazioni principali del cosiddetto *refactoring* (ricomposizione), che consiste nel riscrivere codice già funzionante nell'intento di renderlo più leggibile, più comprensibile e quindi meglio gestibile.¹

Tuttavia, questo desiderio di cambiare e migliorare il codice non è privo di tensioni. Spesso dovete tenere conto di "consumatori", in particolare i programmati-

1. Si veda in proposito *Refactoring: Improving the Design of Existing Code*, di Martin Fowler et al. (Addison-Wesley, 1999). Alcuni programmati disapprovano il refactoring, sostenendo che è inutile rimaneggiare codice che funziona. Il problema di tale approccio è che il ruolo del leone nelle risorse di un progetto, che si tratti di tempo o di denaro, non è rappresentato tanto dalla scrittura del codice iniziale, quanto dal suo mantenimento: è facile comprendere, quindi, come la semplificazione del codice si traduca in denaro sonante.

client, che fanno affidamento sull'invariabilità di alcuni aspetti del vostro codice: voi vorreste modificarlo, mentre essi si aspettano che rimanga immutato. Una delle considerazioni importanti nella progettazione OOP, pertanto, può essere espressa come “separare le cose che cambiano da quelle che rimangono costanti”, ed è fondamentale soprattutto per le librerie.

Gli utenti di una libreria devono poter contare sui componenti che utilizzano e sapere che quando ne uscirà una nuova versione non dovranno riscrivere il codice; d'altro canto, il creatore della libreria deve sentirsi libero di eseguire le modifiche e i miglioramenti che ritiene opportuni, con la certezza che il codice client non verrà influenzato dalle sue modifiche.

Entrambe le esigenze possono essere soddisfatte per mezzo di convenzioni: per esempio, il programmatore di libreria deve evitare di rimuovere metodi esistenti quando modifica una classe della libreria, poiché questo sconvolgerebbe il codice del programmatore client. La situazione inversa è tuttavia più spinosa; nel caso dei campi, come può il creatore della libreria sapere quali sono stati utilizzati dai programmatore client? Questo è vero anche per i metodi che fanno solo parte dell'implementazione di una classe, e non sono ideati per essere adottati direttamente dal programmatore client. E che cosa accadrebbe, poi, se il creatore della libreria volesse liberarsi di una vecchia implementazione per inserirne una nuova? La modifica di un qualunque membro potrebbe alterare il codice del programmatore client. Il creatore della libreria si muove in uno spazio di manovra molto limitato, quindi non può modificare alcunché.

Per risolvere questo problema Java mette a disposizione i cosiddetti *modificatori o specificatori di accesso* (*access specifier*), che permettono al creatore della libreria di precisare ciò che è disponibile al programmatore client e ciò che invece non lo è. I livelli di controllo degli accessi, che vanno da accesso completo a minimo, sono **public**, **protected**, **package access** (che non è indicato da una parola chiave) e **private**.

Da quanto avete appreso nel paragrafo precedente potreste ritener che, come progettista di libreria, sia opportuno rendere “privato” tutto ciò che è possibile, esponendo i soli metodi che volete siano utilizzati dal programmatore client. Questo approccio è corretto, sebbene spesso poco intuitivo per quanti programmano in altri linguaggi, in particolare il C, che abitua a un accesso illimitato. Al termine di questo capitolo dovreste essere convinti del valore del controllo di accesso in Java.

Il concetto di libreria e di controllo su chi può accedere ai suoi componenti non è tuttavia completo, in quanto occorre esaminare il modo per “impacchettare” i vari componenti allo scopo di comporre un’unità di libreria coesa. In Java questo comportamento è controllato tramite la parola chiave

package; i modificatori di accesso sono condizionati dalla posizione di una classe, nello stesso package o in uno diverso. Quindi, per iniziare, vedrete come inserire nei package i componenti di libreria: questo vi consentirà di apprezzare in seguito il significato dei modificatori d’accesso.

Package: l’unità di libreria

Un package contiene un gruppo di classi, organizzate nell’ambito di un unico *spazio di nomi* (*namespace*).

Per esempio, la distribuzione Java standard fornisce una libreria di utilità organizzata nello spazio di nomi **java.util**: una delle classi di **java.util** è chiamata **ArrayList**. Un modo per utilizzare un oggetto **ArrayList** consiste nell’indicare per esteso il nome **java.util.ArrayList**.

```
//: access/FullQualification.java

public class FullQualification {
    public static void main(String[] args) {
        java.util.ArrayList list = new java.util.ArrayList();
    }
} ///:~
```

Questa tecnica diventa rapidamente monotona, quindi è probabile che vorrete sostituirla ben presto con la parola chiave **import**, indicando in questa istruzione l’importazione della classe:

```
//: access/SingleImport.java
import java.util.ArrayList;

public class SingleImport {
    public static void main(String[] args) {
        ArrayList list = new java.util.ArrayList();
    }
} ///:~
```

Ora potete utilizzare la classe **ArrayList** senza dover qualificarla con il nome completo; tuttavia, nessuna delle altre classi contenute in **java.util** è finora



disponibile. Per importare l'intero contenuto del package utilizzate semplicemente il carattere “*”, come avete visto in altri esempi di questo manuale:

```
import java.util.*;
```

La ragione di questo meccanismo d'importazione è fornire uno strumento per gestire gli spazi di nomi. I nomi di tutti i membri delle vostre classi sono isolati l'uno dall'altro: un metodo **f()** presente nella classe **A** non andrà in conflitto con un metodo **f()**, avente la stessa segnatura, disponibile nella classe **B**. Ma che dire dei nomi delle classi? Supponete di avere creato una classe **Stack**, e di averla installata su un sistema che già utilizza una classe **Stack** creata da altri. Questo potenziale conflitto di nomi avviene perché in Java è importante avere il controllo completo sugli spazi di nomi e creare una combinazione identificativa univoca per ogni classe.

La maggior parte degli esempi presentati finora era contenuta in file singoli ed è stata progettata per uso locale, in modo da non dover tenere conto dei nomi dei package. In effetti, però, questi esempi erano contenuti in un package: il cosiddetto *package anonimo o predefinito*. Questa è certamente una possibilità e, per ragioni di chiarezza, sarà la tecnica adottata nel resto del manuale, ognqualvolta sia possibile.

Tuttavia, se prevedete di creare librerie o programmi efficacemente integrabili con altri programmi Java sullo stesso sistema, dovete fare in modo di evitare conflitti con i nomi delle classi.

Un file sorgente Java è normalmente chiamato *unità di compilazione* o talvolta *unità di traduzione*. Ogni unità di compilazione deve avere un nome di file che termina con l'estensione **.java**, in minuscolo, all'interno del quale può esservi una sola classe **public**: questa deve avere lo stesso nome del file, rispettando le lettere maiuscole e minuscole, e deve essere priva di estensione. Ogni unità di compilazione può contenere soltanto una classe **public**: in caso contrario il compilatore segnalera un errore. Se in un'unità di compilazione sono presenti altre classi, esse risulteranno “nascoste” all'esterno del package, in quanto non sono **public** ma presenti solo come supporto alla classe **public** principale.

Organizzazione del codice

Quando compilate un file **.java** ottenete un file di output per ogni classe presente al suo interno; ogni file di output ha il nome di una classe presente nel file **.java**, con l'estensione **.class**: potrete quindi ritrovarvi con un gran numero di file **.class** derivati da pochi file **.java**. Se avete esperienza di



programmazione con un linguaggio compilato sarete certo abituati al fatto che il compilatore genera un modulo intermedio, di solito un file **.obj**, poi assemblato con altri file dello stesso tipo tramite un *linker*, per creare un file eseguibile, o un *librarian*, per creare una libreria. Java funziona in modo diverso: un programma funzionante è composto di un insieme di file **.class** che possono essere assemblati in un unico file JAR (*Java ARchive*) tramite il programma di archiviazione di Java, **jar**. È l'interprete Java che si incarica di trovare, caricare e interpretare questi file.²

Una libreria è l'insieme di questi file **.class**: di solito ogni file sorgente possiede una classe **public** e un numero variabile di classi non **public**, in modo che vi sia un solo componente **public** per ciascun file sorgente. Se volete specificare che tutti questi componenti, ciascuno presente nei propri file **.java** e **.class**, fanno parte di un'unica entità, dovete utilizzare la parola chiave **package**.

Per servirvi di un'istruzione **package** dovete inserirla come prima riga del file sorgente, subito dopo eventuali commenti. Quando specificate

```
package access;
```

indicate che questa unità di compilazione fa parte di una libreria di nome **access**. In altri termini state dicendo che il nome della classe **public** all'interno di questa unità di compilazione è compreso nell'ambito del nome **access**, e che chiunque voglia utilizzare questa classe deve specificarne il nome per intero oppure ricorrere alla parola chiave **import** in combinazione con **access**, secondo le modalità descritte in precedenza. Tenete presente che per convenzione i nomi dei package Java prevedono l'utilizzo delle sole lettere minuscole, anche per le parole intermedie.

Supponete, per esempio, che il nome del file sia **MyClass.java**; ciò significa che in questo file può esistere una sola classe **public** e che il nome di questa classe deve essere **MyClass**, rispettando maiuscole e minuscole:

```
//: access/mypackage/MyClass.java
package access.mypackage;
```

```
public class MyClass {
    // ...
}
```

2. In Java non è obbligatorio utilizzare un interprete: sono disponibili compilatori di codice nativo Java in grado di generare un singolo file eseguibile.

A questo punto, chiunque desideri utilizzare **MyClass** o una qualsiasi delle altre classi **public** presenti in **access** dovrà applicare la parola chiave **import** per rendere disponibili il nome o i nomi presenti in **access**. L'alternativa consiste nel fornire il nome per esteso (*fully qualified name*).

```
//: access/Qualified MyClass.java

public class QualifiedMyClass {
    public static void main(String[] args) {
        access.mypackage.MyClass m =
            new access.mypackage.MyClass();
    }
} //:/~
```

La parola chiave **import** consente di rendere tutto assai più “pulito”.

```
//: access/Imported MyClass.java
import access.mypackage.*;

public class ImportedMyClass {
    public static void main(String[] args) {
        MyClass m = new MyClass();
    }
} //:/~
```

Ricordate che ciò che le parole chiave **package** e **import** permettono di fare come progettisti di librerie, in sintesi, è suddividere un singolo spazio di nomi globale; in questo modo eviterete conflitti, a prescindere da quanti utenti Internet decidano di iniziare a scrivere classi Java.

Come creare nomi di package univoci

Poiché un pacchetto non viene mai veramente “assemblato” in un singolo file, potrebbe essere composto di molti file **.class** che alla fine diventerebbero ingombranti. Per evitare tale inconveniente è opportuno collocare in una sola directory tutti i file **.class** di un determinato pacchetto, vale a dire trarre vantaggio dalla struttura gerarchica del file system del sistema operativo. Questo è uno dei due modi con cui Java affronta il problema del “disordine”; vedrete l'altro metodo a proposito del programma di utilità **jar**.

Raggruppare i file che compongono il package in un'unica sottodirectory risolve anche altri due problemi: la creazione di nomi di pacchetto univoci e la possibilità di trovare classi che potrebbero trovarsi nascoste nell'alberatura di directory. La creazione di nomi di package univoci si ottiene codificando il percorso del file **.class** nel nome del **package**: per convenzione la prima parte del nome del **package** corrisponde all'inverso del dominio Internet appartenente al creatore della classe. Poiché i nomi di dominio Internet sono sicuramente univoci, attenendovi a questa regola il nome del vostro package sarà anch'esso univoco e non dovrete mai affrontare situazioni di conflitto. Esiste ovviamente la possibilità che qualcun altro, registrando il vostro nome di dominio che avrete dimenticato di rinnovare, inizi a scrivere codice Java usando i vostri stessi percorsi, tuttavia si tratta di un'ipotesi piuttosto improbabile. Se non avete un dominio, per creare nomi di pacchetto univoci potreste utilizzare una combinazione alternativa, per esempio il vostro codice fiscale: in ogni caso, se avete deciso di pubblicare codice Java varrà la pena che registrate un nome di dominio Internet.

La seconda parte di questo meccanismo scomponete il nome del **package** ospitato in una directory del vostro sistema, in modo che al momento dell'esecuzione il programma Java che dovesse caricare file **.class** possa individuare la directory in cui questi risiedono.

L'interprete Java opera come segue: per prima cosa recupera la variabile d'ambiente **CLASSPATH** impostata dal sistema operativo, talvolta dal programma d'installazione di Java o da altri strumenti Java che potreste avere installato sul vostro sistema.³

La variabile **CLASSPATH** contiene una o più directory che vengono utilizzate come radici (*root*) nella ricerca dei file **.class**.

Iniziando da questa radice l'interprete prenderà il nome del pacchetto e sostituirà ogni punto con una barra, generando il percorso a partire dalla radice **CLASSPATH**: in pratica il package **foo.bar.baz** diventerà la directory **foo\bar\baz** o **foo/bar/baz**, a seconda del sistema operativo utilizzato. Questo percorso viene poi concatenato alle varie voci presenti nel **CLASSPATH**, e il risultato servirà per cercare il file **.class** con il nome corrispondente alla classe da creare; tenete presente che questa ricerca è eseguita anche in alcune directory standard rispetto alla posizione d'installazione dell'interprete Java.

Per comprendere meglio tale meccanismo, considerate il nome di dominio dell'autore, www.mindview.net. Invertendo questo dominio, net.mindview diventa il nome globale univoco per le classi. Nelle versioni precedenti di Java

³In questo volume i nomi delle variabili d'ambiente figurano in maiuscolo, per esempio **CLASSPATH**.

i suffissi .com, .edu, .org ecc. dovevano essere riportati in maiuscolo, ma questa convenzione è stata modificata in Java 2, pertanto l'intero nome deve essere riportato in minuscolo. In seguito questo nome può essere articolato ulteriormente. Per esempio, se create una libreria chiamata **simple** il nome del pacchetto sarà:

```
package net.mindview.simple;
```

Ora questo nome può essere utilizzato come spazio di nomi di base per i due file seguenti.

```
//: net/mindview/simple/Vector.java
// Creazione di un package.
package net.mindview.simple;

public class Vector {
    public Vector() {
        System.out.println("net.mindview.simple.Vector");
    }
} ///:~
```

Come si è detto, l'istruzione **package** deve essere la prima riga di codice non commentato presente nel file. Il secondo file è molto simile.

```
//: net/mindview/simple/List.java
// Creazione di un package.
package net.mindview.simple;

public class List {
    public List() {
        System.out.println("net.mindview.simple.List");
    }
} ///:~
```

Entrambi i file si trovano nella directory seguente:

```
C:\DOC\JavaT\net\mindview\simple
```

Ricordate che in tutti i file sorgente di questo manuale la prima riga di commento indica la posizione del file nell'alberatura di directory; questa informazione è sfruttata dallo strumento automatico di estrazione del codice utilizzato per creare il manuale.

Leggendo questo percorso al contrario non avrete difficoltà a riconoscere il nome del package **net.mindview.simple**, ma che dire della prima parte del percorso? Questa è gestita tramite la variabile d'ambiente CLASSPATH, che sul sistema dell'autore contiene:

```
CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT
```

Come potete vedere, la variabile CLASSPATH può contenere percorsi di ricerca alternativi.

Tuttavia se utilizzate i file JAR c'è una differenza: nel CLASSPATH dovete indicare anche il nome effettivo del file JAR, non soltanto il suo percorso; quindi, per un file di nome **grape.jar** il CLASSPATH sarebbe analogo a:

```
CLASSPATH=.;D:\JAVA\LIB;C:\flavors\grape.jar
```

Se il CLASSPATH è configurato correttamente, il file seguente può essere collocato in qualsiasi directory:

```
//: access/LibTest.java
// Utilizza la libreria.
import net.mindview.simple.*;

public class LibTest {
    public static void main(String[] args) {
        Vector v = new Vector();
        List l = new List();
    }
} /* Output:
net.mindview.simple.Vector
net.mindview.simple.List
*///:~
```

Quando il compilatore incontra l'istruzione **import** per la libreria **simple**, inizia, nelle directory specificate da CLASSPATH, la ricerca di **net/mindview/simple**, quindi individua i file compilati con i nomi appropriati: **Vector.class**

per la classe **Vector** e **List.class** per **List**. Tenete presente che sia le classi sia i metodi desiderati in **Vector** e in **List** devono essere **public**.

L'impostazione del CLASSPATH è stata una tale prova per i primi utenti di Java (e per l'autore, quando iniziò a utilizzare questo linguaggio), che Sun decise di rielaborarla in modo più elegante nelle successive versioni del JDK. Noterete che, dopo l'installazione, anche non impostando il CLASSPATH sarete comunque in grado di compilare ed eseguire programmi Java di base. Per compilare ed eseguire il codice contenuto nel pacchetto sorgente a corredo di questo manuale, disponibile su www.mindview.net, tuttavia, dovrete aggiungere la directory di base dell'alberatura del codice del manuale al vostro CLASSPATH.

Esercizio 1 (1) Create una classe che fa parte di un package e generate un'istanza della vostra nuova classe all'esterno del package.

Collisioni

Che cosa accade se due librerie vengono importate tramite `**` e includono gli stessi nomi? Per esempio, supponete che un programma esegua queste operazioni:

```
import net.mindview.simple.*;
import java.util.*;
```

Poiché `java.util.*` contiene una classe di nome **Vector** esiste un potenziale pericolo di collisione; tuttavia, finché non scrivete del codice che causa la collisione, non vi sono pericoli. Questo è positivo, perché se così non fosse potreste dover scrivere molto codice per evitare una collisione che forse non avverrà mai.

La collisione si verifica non appena create un oggetto **Vector**:

```
Vector v = new Vector();
```

A quale classe di **Vector** si riferisce il codice? Il compilatore non lo sa e probabilmente neppure chi esamina il codice. Di conseguenza il compilatore inizia a lamentarsi per costringervi a essere esplicativi; per creare il **Vector** Java standard dovete scrivere:

```
java.util.Vector v = new java.util.Vector();
```

Per evitare questo problema, dovete sempre specificare il pacchetto.

Poiché questa sintassi, insieme al CLASSPATH, specifica l'esatta ubicazione di **Vector**, non c'è alcuna necessità dell'istruzione `import java.util.*`, a meno di non dover utilizzare altre classi di `java.util`.

In alternativa potete ricorrere alla sintassi per l'importazione di singole classi, purché non utilizziate entrambi i nomi in conflitto nel medesimo programma, nel qual caso dovrete indicare i nomi per esteso.

Esercizio 2 (1) Usate i frammenti di codice presenti in questo paragrafo e trasformateli in un programma; verificate che non vi siano conflitti.

Librerie personalizzate di strumenti

Con queste conoscenze ora siete in grado di creare librerie di strumenti personalizzate per ridurre o eliminare il codice duplicato. Considerate per esempio l'alias utilizzato in precedenza per `System.out.println()`, allo scopo di ridurre la quantità di codice da scrivere; questo alias potrebbe fare parte di una classe **Print**, in modo da permettervi di realizzare un `import` di tipo **static** nettamente più leggibile:

```
//: net/mindview/util/Print.java
// Metodi Print che possono essere usati senza
// qualificatori, tramite static import di Java SE5:
package net.mindview.util;
import java.io.*;

public class Print {
    // Visualizza con un carattere newline:
    public static void print(Object obj) {
        System.out.println(obj);
    }
    // Visualizza soltanto il carattere newline:
    public static void print() {
        System.out.println();
    }
    // Visualizza senza ritorno a capo:
    public static void printnb(Object obj) {
        System.out.print(obj);
    }
}
```

```
// Il nuovo metodo printf() di Java SE5 (dal linguaggio C):
public static PrintStream
printf(String format, Object... args) {
    return System.out.printf(format, args);
}
} //:~
```

Potete sfruttare questa versione abbreviata della funzionalità di visualizzazione in ogni occasione, sia con il carattere di avanzamento riga (metodo **print()**) sia senza questo carattere (**println()**).

Non è difficile intuire che la posizione di questo file deve trovarsi in una directory che inizia in una di quelle indicate dal CLASSPATH, per proseguire in **net/mindview**. Dopo avere compilato il codice, i metodi **static print()** e **println()** saranno utilizzabili ovunque sul vostro sistema, tramite un'istruzione **import static**.

```
//: access/PrintTest.java
// Utilizza il metodo di visualizzazione statico presente in
// Print.java.
import static net.mindview.util.Print.*;

public class PrintTest {
    public static void main(String[] args) {
        print("Available from now on!");
        print(100);
        print(100L);
        print(3.14159);
    }
} /* Output:
Available from now on!
100
100
3.14159
*//:~
```

Altri componenti di questa libreria potrebbero essere i metodi **range()**, introdotti nel Capitolo 4, che permettono l'utilizzo della sintassi **foreach** nella gestione di semplici sequenze di numeri interi:

```
//: net/mindview/util/Range.java
// Metodi per la creazione di Array che
// possono essere usati senza qualificazione,
// usando static import di Java SE5:
package net.mindview.util;

public class Range {
    // Genera una sequenza [0..n]
    public static int[] range(int n) {
        int[] result = new int[n];
        for(int i = 0; i < n; i++)
            result[i] = i;
        return result;
    }
    // Genera una sequenza [start..end]
    public static int[] range(int start, int end) {
        int sz = end - start;
        int[] result = new int[sz];
        for(int i = 0; i < sz; i++)
            result[i] = start + i;
        return result;
    }
    // Genera una sequenza [start..end] incrementandola di step
    public static int[] range(int start, int end, int step) {
        int sz = (end - start)/step;
        int[] result = new int[sz];
        for(int i = 0; i < sz; i++)
            result[i] = start + (i * step);
        return result;
    }
} //:~
```



D'ora in poi, ogni volta che creerete un nuovo metodo d'utilità valido potrete aggiungerlo alla vostra libreria. Nel prosieguo del volume avrete modo di esaminare altri componenti presenti nella libreria **net.mindview.util**.

Utilizzo di import per modificare i comportamenti

Una funzionalità tipica del linguaggio C che è assente in Java è la *compilazione condizionale*, che consente di ottenere comportamenti diversi semplicemente impostando un apposito parametro, senza modificare altro codice. L'assenza di tale funzionalità in Java è probabilmente dovuta al fatto che in C è utilizzata soprattutto per gestire codice multipiattaforma, compilando porzioni di codice diverse in funzione della piattaforma di destinazione; poiché Java è stato concepito proprio come linguaggio multipiattaforma, questa caratteristica risulta ridondante.

Tuttavia, la compilazione condizionale offre altri campi d'applicazione interessanti: uno dei più comuni è il debug del codice. Le funzionalità di debug vengono di norma abilitate nel corso dello sviluppo del progetto e disattivate nella versione di produzione: per fare questo si modifica il package da importare, correggendo il codice utilizzato nel programma dalla versione di debug a quella di produzione. Tenete presente che questa tecnica può essere impiegata per qualsiasi genere di codice condizionale.

Esercizio 3 (2) Create due pacchetti, **debug** e **debugoff**, contenenti una stessa classe dotata di un metodo **debug()**: la prima versione visualizzerà il proprio argomento **String** a console, mentre la seconda non eseguirà alcunché. Utilizzate un'istruzione **static import** per importare la classe in un programma di test e dimostrate l'effetto della compilazione condizionale.

Osservazioni sui package

È opportuno ricordare che ogni volta che create un package, assegnandogli un nome specificate implicitamente una struttura di directory. Il pacchetto deve trovarsi nella directory indicata dal suo nome, che a sua volta deve essere raggiungibile tramite il CLASSPATH. Eseguire esperimenti con la parola chiave **package** può essere dapprima un po' frustrante, poiché a meno che non vi conformiate alla regola "nome-del-package = percorso-directory" otterrete oscuri messaggi d'errore di runtime relativi all'impossibilità di trovare una certa classe, anche se questa si trova nella stessa directory. Qualora otteneiate messaggi di questo tipo, provate a commentare l'istruzione **package**: se il codice compilerà senza ostacoli avrete individuato il problema.



Noteate che il codice compilato si trova spesso in directory diverse da quella del codice sorgente: in ogni caso, il percorso del codice compilato deve essere accessibile alla JVM tramite il CLASSPATH.

Modificatori di accesso in Java

I modificatori di accesso utilizzati in Java, **public**, **protected** e **private**, vengono anteposti alla definizione di ogni membro della classe, sia esso un campo o un metodo. Ogni modificatore, infatti, controlla l'accesso della specifica definizione che lo segue.

Non specificando un modificatore è come se indicaste "package access": pertanto, in un modo o nell'altro, tutto è soggetto a un controllo d'accesso. Nei prossimi paragrafi potrete esaminare in dettaglio i diversi tipi di accesso.

Package access

In nessuno degli esempi presentati prima di questo capitolo si è fatto uso dei modificatori d'accesso: l'accesso predefinito, infatti, non è indicato da nessuna parola chiave, ma è comunemente chiamato *package access* (accesso di pacchetto) o talvolta *friendly* (accesso amichevole). Questo tipo di accesso indica che tutte le altre classi contenute nel package corrente possono accedere al membro, mentre per tutte le classi esterne al package tale membro risulterà **private**. Dal momento che un'unità di compilazione, ossia un file, può appartenere a un solo package, tutte le classi interne a una determinata unità di compilazione sono automaticamente disponibili l'una all'altra tramite l'accesso di pacchetto.

La modalità package access permette di raggruppare in uno stesso pacchetto classi tra loro correlate, affinché possano interagire facilmente le une con le altre. Il raggruppamento di varie classi in un package, garantendo accesso reciproco ai loro membri, si definisce in gergo con l'espressione "possedere il codice di un package". Chiaramente, soltanto il codice che possedete dovrebbe disporre dell'accesso di pacchetto nei confronti di altro codice di cui siete proprietari. Si potrebbe affermare che il package access dà significato al raggruppamento di classi in un package. Molti linguaggi ammettono una certa libertà nel modo di organizzare le definizioni in file, tuttavia Java costringe a organizzarle in modo sensato e rende opportuna l'esclusione di classi che non dovrebbero avere accesso a quelle definite nel package corrente.

La classe esegue controlli sul codice che può accedere ai suoi membri: non basta che il codice di un altro package si presenti con un "Ciao, sono un ami-

co di Marco” perché gli vengano resi disponibili i membri **protected**, **package-access** e **private** di Marco!

L’accesso a un membro è garantito unicamente dal soddisfacimento delle condizioni elencate di seguito.

1. Rendere il membro **public**, in modo che chiunque possa accedervi da qualsiasi punto.
2. Assegnare al membro un accesso di pacchetto, semplicemente non specificando alcun modificatore di accesso, e poi collocare le altre classi nello stesso package: così facendo tutte le altre classi del pacchetto potranno accedere al membro.
3. Come vedrete nel prossimo capitolo, in cui sarà introdotta l’ereditarietà, una classe ereditata può avere accesso ai membri **protected** e **public**, ma non ai membri **private**; può accedere a un membro con package access soltanto se le due classi risiedono nello stesso pacchetto. Tuttavia, per il momento non preoccupatevi dell’ereditarietà e dell’accesso **protected**.
4. Fornire metodi “accessor/mutator”, detti anche metodi “get/set”, che leggono e modificano i valori. In termini di programmazione a oggetti questo è l’approccio consigliato e, come vedrete nel Volume 3, Capitolo 2, dedicato alle interfacce grafiche, fondamentale in JavaBeans.

public: accesso di interfaccia

Specificando la parola chiave **public** fate sì che il membro la cui dichiarazione segue immediatamente questa parola sia disponibile a tutti, in particolare al programmatore client che utilizza la libreria. Supponete di definire un package chiamato **dessert**, contenente l’unità di compilazione definita di seguito.

```
//: access/dessert/Cookie.java
// Crea una libreria.
package access.dessert;

public class Cookie {
    public Cookie() {
        System.out.println("Cookie constructor");
    }
    void bite() { System.out.println("bite"); }
} ///:~
```

Ricordate che il file di classe prodotto da **Cookie.java** deve risiedere nella sottodirectory **dessert** di una directory chiamata **access**, che a sua volta deve trovarsi in una delle directory specificate in **CLASSPATH**. Non commettete l’errore di ritenere che Java utilizzerà sempre la directory corrente come uno dei punti di partenza delle sue ricerche di percorso: se nel vostro **CLASSPATH** non avete specificato anche un “.”, che indica universalmente la directory corrente, Java non ne terrà conto.

Ora, se create un programma che utilizza **Cookie**:

```
//: access/Dinner.java
// Usa la libreria.
import access.dessert.*;

public class Dinner {
    public static void main(String[] args) {
        Cookie x = new Cookie();
        //! x.bite(); // Accesso impossibile
    }
} /* Output:
Cookie constructor
*///:~
```

potrete generare un oggetto **Cookie**, perché sia il suo costruttore sia la classe sono **public**: nel prosieguo del libro vedrete meglio il concetto di classe **public**. Tuttavia il membro **bite()** non è accessibile dall’interno di **Dinner.java**, perché **bite()** garantisce l’accesso solo nell’ambito del package **dessert**: pertanto il compilatore vi impedirà di utilizzarlo.

Package predefinito

Potrebbe sorprendervi scoprire che il codice seguente compila regolarmente, sebbene all’apparenza violi le regole. Il codice è strutturato in un primo file:

```
//: access/Cake.java
// Accede a una classe in un’unita’ di compilazione separata
class Cake {
    public static void main(String[] args) {
        Pie x = new Pie();
        x.f();
    }
}
```



```

}
} /* Output:
Pie.f()
*/:~
```

e in un secondo file, nella stessa directory:

```

//: access/Pie.java
// L'altra classe.

class Pie {
    void f() { System.out.println("Pie.f()"); }
} */:~
```

A una prima occhiata potreste considerare che questi file sono del tutto estranei, tuttavia **Cake** è in grado di creare un oggetto **Pie** e chiamare il suo metodo **f()**; ricordate che per compilare questo esempio il vostro CLASSPATH deve contenere “.”. Di norma sareste portati a credere che **Pie** e **f()** sono caratterizzati dall'accesso di pacchetto e che, per questo, non sono disponibili a **Cake**. In effetti la prima parte di questa assunzione è corretta; tuttavia, sia l'oggetto **Pie** sia il metodo **f()** sono invece disponibili a **Cake.java** perché a) si trovano nella stessa directory e b) non è stato esplicitamente definito alcun nome di pacchetto. Java considera i file di questo tipo come implicitamente appartenenti a un “pacchetto predefinito” di quella directory, il cosiddetto *default package*, e per questo motivo garantisce un accesso di tipo package access a tutti gli altri file presenti nella stessa directory.

private: proibito toccare!

La parola chiave **private** indica che nessuno può accedere a un determinato membro tranne la classe che lo possiede, e soltanto dai propri metodi: altre classi nello stesso package non possono accedere ai membri **private**, quindi è come se la classe fosse “protetta”, anche da chi l'ha creata. Non è raro, infatti, che un package sia realizzato in collaborazione tra vari programmatore; in tal caso **private** consente di gestire a vostro piacimento quel membro senza preoccuparvi dei suoi possibili effetti su altre classi dello stesso pacchetto.

Generalmente l'accesso predefinito di tipo package access offre caratteristiche di occultamento adeguate a un utilizzo generico: ricordate che un membro con accesso di pacchetto è inaccessibile al programmatore client che si



serve della classe. Poiché l'accesso predefinito è quello normalmente adottato, si tratta di un'eccellente funzionalità, dal momento che per ottenerla non è necessario specificare alcun modificatore di accesso.

Quindi, di norma dovete tenere conto soltanto degli accessi per i membri che vorrete esplicitamente rendere **public** al programmatore client. Almeno agli inizi, è probabile che non utilizzerete spesso la parola chiave **private**, tenuto conto che il suo utilizzo non è indispensabile; tuttavia un uso coerente di **private** è requisito fondamentale soprattutto nel campo del multithreading, come vedrete nel Volume 3, Capitolo 1 dedicato alla concorrenza.

Di seguito è mostrato un esempio di utilizzo del modificatore **private**.

```

//: access/IceCream.java
// Dimostra l'uso della parola chiave "private".
```

```

class Sundae {
    private Sundae() {}
    static Sundae makeASundae() {
        return new Sundae();
    }
}

public class IceCream {
    public static void main(String[] args) {
        //! Sundae x = new Sundae();
        Sundae x = Sundae.makeASundae();
    }
} */:~
```

Questo codice mostra un tipico caso in cui **private** si rivela utile: la possibilità di controllare come viene creato un oggetto e impedire che qualcuno acceda direttamente a uno o a tutti i costruttori. Nell'esempio precedente non potete creare un oggetto **Sundae** per mezzo del suo costruttore, ma dovete chiamare il metodo **makeASundae()**, che si occupa di farlo.⁴

4. In questo caso è messo in evidenza un altro effetto: poiché il costruttore predefinito è l'unico a essere stato definito ed è **private**, impedisce l'applicazione del principio di ereditarietà, un argomento che vedrete in seguito.



Potete rendere **private** i metodi che siete certi abbiano unicamente funzione di “supporto” a una classe (metodi *helper*), per essere sicuri di non utilizzarli inavvertitamente in altri punti del package.

Questo è vero anche per un campo **private** della classe: a meno di dover esporre l’implementazione, evento meno probabile di quanto potreste ritenere, dovreste rendere tutti i campi **private**. Tuttavia, il semplice fatto che un riferimento a un oggetto sia **private** all’interno di una classe non significa che un altro oggetto non possa avere un riferimento **public** allo stesso oggetto. Per quanto riguarda il trattamento degli alias, consultate i riferimenti online di questo manuale.

protected: accesso per ereditarietà

Non è fondamentale comprendere i concetti illustrati in questo paragrafo per continuare la lettura fino all’ereditarietà, che vedrete nel Capitolo 7; per completezza, in ogni caso, si è ritenuto utile descrivere brevemente e illustrare con un esempio l’utilizzo di **protected**.

La parola chiave **protected** rientra in un concetto chiamato ereditarietà: si crea una nuova classe che ne duplica una esistente, chiamata superclasse o classe di base; alla nuova classe si aggiungono nuovi membri o si modifica il comportamento di quelli “ereditati” dalla classe originale. Per ereditare da una classe basta indicare che quella nuova “amplia” (**extends**) quella esistente, come in:

```
class Foo extends Bar {
```

Il resto della definizione della classe rimane invariato.

Se generate un nuovo package ed ereditate da una classe presente in un altro package, gli unici membri cui avrete accesso sono quelli **public** del pacchetto originale; naturalmente, qualora applicaste l’ereditarietà nello stesso package potreste gestire tutti i membri che sono caratterizzati dall’accesso package access. Talvolta il creatore della classe di base vorrebbe consentire l’accesso a un determinato membro solo da parte delle classi derivate, non da parte di chiunque. Questo è lo scopo del modificatore **protected**; esso garantisce anche il package access, ossia la possibilità che altre classi dello stesso package accedano agli elementi **protected**.



Per ritornare al file **Cookie.java**, nell’esempio seguente la classe non può chiamare il membro **bite()**, che dispone di package access.

```
//: access/ChocolateChip.java
/* Non si possono usare membri package-access da un altro
   package.*/
import access.dessert.*;

public class ChocolateChip extends Cookie {
    public ChocolateChip() {
        System.out.println("ChocolateChip constructor");
    }
    public void chomp() {
        //! bite(); // Impossibile accedere al metodo bite
    }
    public static void main(String[] args) {
        ChocolateChip x = new ChocolateChip();
        x.chomp();
    }
} /* Output:
Cookie constructor
ChocolateChip constructor
*//*:~
```

Una delle caratteristiche interessanti dell’ereditarietà è che se il metodo **bite()** esiste nella classe **Cookie**, allora esiste anche in qualsiasi classe abbia ereditato da **Cookie**. Tuttavia, dal momento che **bite()** è dotata di package access e si trova in un package esterno, non è disponibile in questo pacchetto. Certamente potreste rendere il metodo **public**, ma in tal caso chiunque potrebbe accedervi, e questo potrebbe non essere il comportamento ideale. Se modificate la classe **Cookie** nel modo seguente:

```
//: access/cookie2/Cookie.java
package access.cookie2;

public class Cookie {
    public Cookie() {
        System.out.println("Cookie constructor");
```



```

}
protected void bite() {
    System.out.println("bite");
}
} //:~

```

il metodo **bite()** diventerà accessibile a chiunque erediti da **Cookie**:

```

//: access/ChocolateChip2.java
import access.cookie2.*;

public class ChocolateChip2 extends Cookie {
    public ChocolateChip2() {
        System.out.println("ChocolateChip2 constructor");
    }
    public void chomp() { bite(); } // Metodo Protected
    public static void main(String[] args) {
        ChocolateChip2 x = new ChocolateChip2();
        x.chomp();
    }
} /* Output:
Cookie constructor
ChocolateChip2 constructor
bite
*//:~

```

Ricordate che malgrado il metodo **bite()** abbia un accesso di tipo package access, non è **public**.

Esercizio 4 (2) Dimostrate che i metodi **protected** hanno package access ma non sono **public**.

Esercizio 5 (2) Create una classe con campi e metodi di tipo **public**, **private**, **protected** e **package-access**; generate un oggetto di questa classe e osservate i messaggi generati dal compilatore quando tentate di accedere a tutti i membri della classe. Tenete presente che le classi nella stessa directory appartengono al package “predefinito”.



Esercizio 6 (1) Create una classe contenente dati di tipo **protected**; nello stesso file, create poi una seconda classe con un metodo che manipola i dati **protected** della prima.

Interfaccia e implementazione

Il controllo di accesso viene spesso definito con l'espressione “occultamento dell'implementazione” (*implementation hiding*). L'azione combinata di inglobare i dati e i metodi nelle classi e nascondere l'implementazione genera il cosiddetto encapsulamento: il risultato è un tipo di dato con caratteristiche e comportamenti propri.⁵

Il controllo di accesso pone limiti al tipo di dati, per due motivi importanti. Il primo è stabilire ciò che ai programmatore client è permesso utilizzare: potete quindi incorporare i vostri meccanismi interni nella struttura, senza preoccuparvi che i programmatore client accedano casualmente a questa implementazione e la gestiscano come parte dell'interfaccia che dovrebbero utilizzare.

Questo conduce direttamente al secondo motivo, vale a dire la possibilità di separare l'interfaccia dall'implementazione. Immaginate che la struttura sia usata in un certo numero di programmi: se ai programmatore client non è consentito esercitare l’“onnipotenza”, ma soltanto inviare messaggi all'interfaccia **public**, voi sarete liberi di modificare qualunque elemento che non sia **public**, cioè i membri con package access, **protected** o **private**, senza interferire con il codice del programmatore client.

Per chiarezza potreste adottare uno stile di programmazione che prevede la creazione di classi in cui si indicano per primi tutti i membri **public**, seguiti dai **protected**, **package-access** e infine **private**. Il vantaggio di tale approccio risiede nel fatto che l'utente della classe, analizzando il codice, trova all'inizio ciò che gli interessa: i membri **public**, i soli che possa utilizzare. L'utente potrà poi interrompere la lettura del codice non appena incontrerà membri non **public**, che appartengono all'implementazione interna.

```
//: access/OrganizedByAccess.java
```

```
public class OrganizedByAccess {
    public void pub1() { /* ... */ }
```

5. Tenete presente che spesso il termine *encapsulamento* viene utilizzato impropriamente per indicare il solo occultamento dell'implementazione.

```
public void pub2() { /* ... */ }
public void pub3() { /* ... */ }
private void priv1() { /* ... */ }
private void priv2() { /* ... */ }
private void priv3() { /* ... */ }
private int i;
// ...
} // :~
```

Purtroppo questa tecnica si limiterà a rendere il codice un po' più facile da consultare, poiché in ogni caso l'interfaccia e l'implementazione sono ancora insieme: in pratica, il codice sorgente (l'implementazione) è ancora visibile poiché contenuto nella classe.

Tra l'altro, occorre tenere conto che i commenti di documentazione supportati da Javadoc ridimensionano l'importanza della leggibilità del codice da parte del programmatore client: il compito di visualizzare l'interfaccia disponibile all'utente di una classe viene delegato al browser di classi (*class browser*), un pratico strumento che mostra le classi disponibili e quello che è consentito farne, ovvero i membri utilizzabili. In Java la visualizzazione della documentazione JDK mediante un browser web offre la stessa funzionalità del browser di classi.

Accesso alle classi

Java consente di utilizzare i modificatori d'accesso anche per determinare quali classi contenute all'interno di una libreria sono disponibili agli utenti di quella stessa libreria. Se desiderate che una classe sia fruibile da parte di un programmatore client, utilizzate la parola chiave **public** sull'intera definizione di classe: in tal modo potrete controllare anche se il programmatore client può creare un oggetto di quella classe.

Per controllare l'accesso a una classe il modificatore deve essere posto prima della parola chiave **class**, come nell'esempio seguente:

```
public class Widget {
```

Ora, se il nome della vostra libreria è **access**, qualsiasi programmatore client potrà accedere a **Widget** scrivendo nel proprio codice:

```
import access.Widget;
```

oppure

```
import access.*;
```

Questa tecnica implica una serie di vincoli aggiuntivi.

1. Per ogni unità di compilazione (file) può essere presente una sola classe **public**: il concetto è che ogni unità possiede una sola interfaccia pubblica, rappresentata da questa classe **public**. Può essere presente invece un numero qualsiasi di classi di supporto con package access. Se un file contiene più di una classe **public** il compilatore segnalerà un errore.
2. Il nome della classe **public** deve corrispondere esattamente al nome del file che contiene l'unità di compilazione: pertanto, per la classe **Widget** il nome del file deve essere **Widget.java**, non **widget.java** né **WIDGET.java**. Se così non fosse otterreste un errore di compilazione.
3. Malgrado non sia pratica comune, nulla vieta di avere un'unità di compilazione che non contiene classi **public**. In tal caso potete assegnare al file il nome che preferite; ricordate comunque che un nome arbitrario è spesso fuorviante per chi legge e gestisce il codice.

E se in **access** fosse presente una classe utilizzata esclusivamente per supportare le operazioni di **Widget** o di altre classi **public** di **access**?

In questo caso non vorrete certo affrontare l'inconveniente di creare documentazione per il programmatore client; d'altro canto, prima o poi potrete decidere di modificare il funzionamento di questa classe di supporto o sostituirla con una diversa. Per usufruire di questa flessibilità, tuttavia, dovete accertarvi che nessun programmatore client dipenda dai dettagli di implementazione nascosti in **access**, in questo caso specifico dalla classe di supporto. A questo scopo non dovete fare altro che togliere la parola chiave **public** dalla definizione della classe, che assumerà così il tipo di accesso package access, tornando a essere utilizzabile soltanto all'interno del package.

Esercizio 7 (1) Create una libreria basandovi sui frammenti di codice che descrivono **access** e **Widget**. Create un oggetto **Widget** in una classe non appartenente al package chiamato **access**.

Quando si crea una classe con accesso di pacchetto, potrebbe avere senso definire i campi come **private**; in effetti, quando possibile dovreste sempre farlo. Generalmente, comunque, è altrettanto ragionevole fornire ai metodi lo stesso tipo di accesso della classe (package-access). Poiché di solito una classe package-access è utilizzata unicamente all'interno del pacchetto, sarà sufficiente creare come **public** soltanto quei metodi di questa classe che il compilatore segnalerà come indispensabili.

Tenete presente che una classe non può essere né **private**, poiché ciò la renderebbe inaccessibile a chiunque tranne alla classe stessa, né **protected**.⁶

Di conseguenza per le classi sono possibili solo due scelte: package access o **public**. Se volete che nessun altro abbia accesso a una classe, potete renderne **private** anche i costruttori: in questo modo, all'interno di un membro **static** della classe soltanto voi potrete creare un oggetto di quella classe. Considerate l'esempio seguente:

```
//: access/Lunch.java
// Dimostra l'uso dei modificatori di accesso per le classi,
// creando una classe private con costruttori private

class Soup1 {
    private Soup1() {}
    // (1) Consente la creazione tramite un metodo static:
    public static Soup1 makeSoup() {
        return new Soup1();
    }
}

class Soup2 {
    private Soup2() {}
    // (2) Crea un oggetto static e restituisce un riferimento
    // a richiesta, secondo il pattern "Singleton":
    private static Soup2 ps1 = new Soup2();
    public static Soup2 access() {
        return ps1;
    }
    public void fo() {}
}

// Puo' esserci una sola classe public per ogni file:
public class Lunch {
    void testPrivate() {
        // Non si puo' fare perchc il costruttore e' private:
        //! Soup1 soup = new Soup1();
    }
}
```

6. In realtà esiste il caso speciale delle classi interne (*inner class*), che possono essere **private** o **protected**: l'argomento sarà illustrato nel Capitolo 10.

```
// Non si puo' fare perchc il costruttore e' private:
//! Soup1 soup = new Soup1();
}
void testStatic() {
    Soup1 soup = Soup1.makeSoup();
}
void testSingleton() {
    Soup2.access().f();
}
} //:~
```

Finora la maggior parte dei metodi ha restituito **void** o un tipo primitivo, quindi la definizione:

```
public static Soup1 makeSoup() {
    return new Soup1();
}
```

potrebbe sembrare a prima vista fuorviante. Il termine **Soup1** prima del nome **makeSoup()** indica ciò che viene restituito da questo metodo. In questo manuale si è sempre fatto uso di **void**, per indicare che il metodo non restituisce alcunché; vedete, però, che è anche possibile restituire un riferimento a un oggetto, come in questo caso in cui il metodo restituisce un riferimento a un oggetto della classe **Soup1**.

Le classi **Soup1** e **Soup2** mostrano come evitare la creazione diretta di una classe, rendendo **private** tutti i costruttori. Ricordate che se non create esplicitamente almeno un costruttore, Java ne genererà uno predefinito, senza argomenti; di contro, scrivendo il costruttore predefinito esso non verrà creato automaticamente. Rendendolo **private**, inoltre, nessuno potrà creare un oggetto di quella classe. Ma allora, come può un utente utilizzare questa classe? L'esempio precedente illustra due opzioni, descritte di seguito.

Nella prima, in **Soup1** viene creato un metodo **static** che genera un nuovo **Soup1** e ne restituisce il riferimento; questa tecnica può esservi utile se dovete eseguire operazioni supplementari con **Soup1** prima che venga restituito, o qualora desideriate mantenere il conteggio degli oggetti **Soup1** creati, eventualmente per limitarne il numero.

Nella seconda opzione **Soup2** ricorre al cosiddetto *design pattern*, argomento trattato in modo approfondito in *Thinking in Patterns (with Java)* disponibile su www.mindview.net: questo schema particolare è chiamato *Singleton*.



poiché permette la creazione di un solo oggetto. L'oggetto della classe **Soup2** viene generato come membro **static private** di **Soup2**: pertanto può esistere uno soltanto, ottenibile unicamente tramite il metodo **public** chiamato **access()**.

Come sapete, non fornendo un modificatore d'accesso per la classe si ottiene automaticamente il package access: questo significa che un oggetto della classe in esame può essere creato da qualsiasi altra classe presente nel package ma non da classi esterne a esso. Ricordate, infatti, che tutti i file presenti nella stessa directory che sono privi di un'istruzione **package** esplicita appartengono implicitamente al package predefinito per la directory. Tuttavia se un membro **static** di questa classe è **public** il programmatore client potrà ancora accedervi, pur non potendo creare un oggetto di questa classe.

Esercizio 8 (4) Sull'esempio di **Lunch.java** create una classe **ConnectionManager** che gestisce un array fisso di oggetti **Connection**. Al programmatore client non dovrà essere possibile creare esplicitamente oggetti **Connection**, ma potrà ottenerli solo per mezzo di un metodo **static** di **ConnectionManager**; qualora il **ConnectionManager** esaurisca gli oggetti, restituirà un riferimento nullo. Testate le classi nel metodo **main()**.

Esercizio 9 (2) Create il file seguente nella directory **access/local**, che dovrebbe trovarsi nel vostro CLASSPATH.

```
// access/local/PackagedClass.java
package access.local;

class PackagedClass {
    public PackagedClass() {
        System.out.println("Creating a packaged class");
    }
}
```

Poi create il seguente file in una directory diversa da **access/local**.

```
// access/foreign/Foreign.java
package access.foreign;
import access.local.*;

public class Foreign {
```



```
public static void main(String[] args) {
    PackagedClass pc = new PackagedClass();
}
```

Spiegate perché il compilatore genera un errore. Che cosa cambierebbe rendendo la classe **Foreign** parte del package **access.local**?

Riepilogo

In ogni relazione è importante che esistano limiti rispettati da tutte le parti coinvolte. Quando create una libreria, stabilite una relazione con il suo utente, il programmatore client, che la utilizza per costruire applicazioni o altre librerie.

In assenza di regole i programmatore client possono fare pressoché qualsiasi cosa desiderino con i membri di una classe, anche se vorreste che alcuni membri non fossero direttamente manipolabili.

In questo capitolo avete visto come combinare le classi per costruire librerie: in primo luogo come assemblare un gruppo di classi in una libreria, poi come la classe controlla l'accesso ai propri membri.

È stato stimato che un progetto realizzato in C inizia a rovinarsi tra le 50.000 e le 100.000 righe di codice: questo avviene perché C offre un solo spazio di nomi e questi iniziano a collidere, determinando un onere aggiuntivo per la gestione di questi problemi. In Java, la parola chiave **package**, lo schema di nomenclatura del package e la parola chiave **import** assicurano il controllo completo sui nomi e consentono di evitare il problema delle collisioni.

Le ragioni per controllare l'accesso a membri sono due: la prima è evitare che gli utenti client accedano a porzioni di codice che non dovrebbero toccare, indispensabili per l'esecuzione di operazioni interne della classe, ma che non appartengono all'interfaccia di cui il programmatore client necessita. Rendendo **private** i metodi e i campi offrite anche un servizio ai programmatore client, che riescono così a individuare quanto è importante per la loro operatività e quanto possono invece ignorare, e facilitate loro la comprensione della classe.

La seconda ragione per controllare l'accesso, ben più importante, è consentire al progettista della libreria di modificare i meccanismi interni della classe senza preoccuparsi degli effetti che il suo lavoro potrebbe avere su quello del programmatore client. Per esempio, potrete costruire una classe in un certo



modo e scoprire poi che ristrutturando il codice otterrete una maggiore velocità d'esecuzione: se l'interfaccia e l'implementazione sono ben separate e protette, potete procedere a qualsiasi intervento senza costringere i programmati client a riscrivere il loro codice. Il controllo dell'accesso assicura che nessun programmatore client dipenda da un componente qualsiasi dell'implementazione di una classe.

Quando avete la possibilità di modificare l'implementazione sottostante, non soltanto disponete della libertà di perfezionare il vostro progetto, ma potete anche permettervi di commettere errori: ricordate che, per quanto la progettazione venga pianificata con cura, gli errori sono sempre in agguato. Il fatto di sapere di poter sbagliare senza compromettere aspetti essenziali del progetto consente di sperimentare meglio, di apprendere più in fretta e di completare il progetto in tempi minori.

L'interfaccia pubblica di una classe è ciò che l'utente client vedrà, pertanto è il componente cui si deve prestare maggiore attenzione in fase di analisi e progettazione. Anche in questo caso avete margini di manovra ragionevoli: se non ottenete l'interfaccia perfetta al primo colpo potrete aggiungere altri metodi, purché non rimuoviate uno qualsiasi di quelli che i programmati client hanno già impiegato nel loro codice.

Tenete presente che il controllo di accesso si focalizza sulla relazione e su una sorta di "comunicazione" tra il creatore della libreria e i client esterni di questa libreria.

Tuttavia esistono molti casi in cui questo non avviene, per esempio quando scrivete il codice da soli o se lavorate con un piccolo gruppo di programmati e tutto il codice è parte dello stesso package.

Tali situazioni richiedono un diverso tipo di comunicazione e la rigida aderenza alle regole potrebbe essere inadatta: in questi casi l'accesso predefinito potrebbe essere la soluzione più opportuna.

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

Il codice di questo capitolo è disponibile sul sito www.mindview.net.
Per scaricarlo, cliccate su "Capitoli" e poi su "Capitolo 7".
L'elenco dei file disponibili include:
• *Cap07.java*: il codice sorgente principale.
• *Cloud.java*: un file di testo contenente i dati per la classe *Cloud*.
• *Clouds.java*: un file di testo contenente i dati per la classe *Clouds*.
• *CloudsTest.java*: il codice sorgente per la classe *CloudsTest*.

Capitolo 7

Riutilizzo delle classi



Una delle caratteristiche più irresistibili di Java è la possibilità di riutilizzo del codice: ma per essere rivoluzionari non basta copiare codice e modificarlo.

Questo è l'approccio adottato nei linguaggi procedurali come C e non si è rivelato vincente. Come tutto in Java, la soluzione ruota intorno al concetto di classe: riutilizzate il codice creando nuove classi, ma anziché crearle *ex-novo* vi servite di classi esistenti che qualcuno ha già costruito e sottoposto a debugging.

Il trucco consiste nell'utilizzare le classi senza "sporcare" il codice esistente, e in questo capitolo vedrete due modi per farlo. Il primo è immediato, visto che non dovete fare altro che creare oggetti della vostra classe dentro la nuova classe. Questo approccio è chiamato *composizione*, poiché la nuova classe è "composta" di oggetti di classi esistenti: riutilizzate semplicemente le funzionalità del codice, non la sua forma.

Il secondo approccio è più ingegnoso e prevede di creare una nuova classe come un tipo di una esistente: consist senza modificarla. Questa tecnica è chiamata *ereditarietà*, e la sua esecuzione è quasi interamente a carico del compilatore. Considerata una delle pietre di volta della programmazione a oggetti, l'ereditarietà ha implicazioni aggiuntive che vedrete in dettaglio nel Capitolo 8.



Molti comportamenti e sintassi sono comuni sia alla composizione sia all'ereditarietà: in effetti, questo ha senso poiché entrambe sono tecniche per ottenere nuovi tipi da tipi esistenti. In questo capitolo apprenderete l'uso di questi meccanismi di riutilizzo del codice.

Sintassi della composizione

In questo manuale la composizione è già stata utilizzata in numerose occasioni, e richiede semplicemente di **porre riferimenti di oggetto all'interno di nuove classi**. Per esempio, supponete di volere creare un oggetto contenente diversi oggetti **String**, un paio di primitivi e un oggetto di un'altra classe. Per gli oggetti non primitivi inserirete riferimenti nella vostra nuova classe, mentre definirete direttamente quelli primitivi.

```
//: reusing/SprinklerSystem.java
// Composizione per riutilizzo del codice.

class WaterSource {
    private String s;
    WaterSource() {
        System.out.println("WaterSource()");
        s = "Constructed";
    }
    public String toString() { return s; }
}

public class SprinklerSystem {
    private String valve1, valve2, valve3, valve4;
    private WaterSource source = new WaterSource();
    private int i;
    private float f;
    public String toString() {
        return
            "valve1 = " + valve1 + " " +
            "valve2 = " + valve2 + " " +
            "valve3 = " + valve3 + " " +
            "valve4 = " + valve4 + "\n" +
            "source = " + source;
    }
}
```



```
"i = " + i + " " + "f = " + f + " " +
"source = " + source;
}

public static void main(String[] args) {
    SprinklerSystem sprinklers = new SprinklerSystem();
    System.out.println(sprinklers);
}

} /* Output:
WaterSource()
valve1 = null valve2 = null valve3 = null valve4 = null
i = 0 f = 0.0 source = Constructed
*///:~
```

Uno dei metodi definiti in entrambe le classi è speciale: **toString()**. Ogni oggetto non primitivo ha un metodo **toString()**, chiamato in occasioni particolari quando il compilatore vuole ottenere una **String** pur disponendo di un oggetto. Quindi, nell'espressione in **SprinklerSystem.toString()**

```
"source = " + source;
```

il compilatore rileva che state cercando di aggiungere un oggetto **String** ("source =") a una sorgente d'acqua, **WaterSource**. Dal momento che Java permette soltanto di "aggiungere" una **String** a un'altra, il compilatore si incarica di trasformare **source** in una **String** chiamando **toString()**. Fatto questo, può concatenare le due stringhe e passare la **String** risultante a **System.out.println()**, oppure agli equivalenti metodi **static** di questo libro, **print()** e **println()**. Ricordate che ogniqualvolta vogliate riprodurre questo comportamento in una classe sarà sufficiente che scriviate un metodo **toString()**.

Come avete visto nel Capitolo 2, i primitivi che sono campi di una classe vengono inizializzati automaticamente a zero. Tuttavia i riferimenti di oggetto vengono inizializzati a **null**, e se provate a chiamare metodi per uno qualunque di essi otterrete un errore di runtime, la cosiddetta eccezione. Grazie al metodo **toString()** potete visualizzare un riferimento **null** senza produrre un'eccezione.

È del tutto comprensibile che il compilatore non si limiti a creare un oggetto predefinito per ogni riferimento, poiché questo comporterebbe nella maggior parte dei casi un inutile incremento nel carico di lavoro. Se volete che i riferimenti siano inizializzati potete farlo esplicitamente.

1. Al momento di definire gli oggetti: questo significa che essi saranno sempre inizializzati prima della chiamata al costruttore.
2. Nel costruttore della classe.
3. Subito prima di utilizzare l'oggetto, secondo la cosiddetta *lazy initialization* (inizializzazione pigra): questa tecnica contribuisce a ridurre il carico di lavoro nei casi in cui la creazione dell'oggetto sia onerosa e qualora l'oggetto non debba essere creato ogni volta.
4. Mediante l'inizializzazione dell'istanza.

I quattro approcci sono illustrati nel codice seguente:

```
//: reusing/Bath.java
// Inizializzazione del costruttore con composizione.
import static net.mindview.util.Print.*;

class Soap {
    private String s;
    Soap() {
        print("Soap()");
        s = "Constructed";
    }
    public String toString() { return s; }
}

public class Bath {
    private String // Inizializzazione al punto di definizione:
    s1 = "Happy",
    s2 = "Happy",
    s3, s4;
    private Soap castille;
    private int i;
    private float toy;
    public Bath() {
        print("Inside Bath()");
        s3 = "Joy";
        toy = 3.14f;
        castille = new Soap();
    }
}
```

```
}
// Inizializzazione dell'istanza:
{ i = 47; }
public String toString() {
    if(s4 == null) // Inizializzazione differita:
        s4 = "Joy";
    return
        "s1 = " + s1 + "\n" +
        "s2 = " + s2 + "\n" +
        "s3 = " + s3 + "\n" +
        "s4 = " + s4 + "\n" +
        "i = " + i + "\n" +
        "toy = " + toy + "\n" +
        "castille = " + castille;
}
public static void main(String[] args) {
    Bath b = new Bath();
    print(b);
}
} /* Output:
Inside Bath()
Soap()
s1 = Happy
s2 = Happy
s3 = Joy
s4 = Joy
i = 47
toy = 3.14
castille = Constructed
*///:~
```

Osservate che nel costruttore **Bath** viene eseguita un'istruzione prima che abbia luogo una qualunque delle inizializzazioni. Se non eseguite l'inizializzazione al momento della definizione, non avrete alcuna garanzia che avvenga un qualsiasi tipo d'inizializzazione prima di inviare un messaggio al riferimento di un oggetto, se non perché verrà prodotta l'inevitabile eccezione in fase di esecuzione.

La chiamata a `toString()` popola il riferimento `s4`, in modo che tutti i campi siano correttamente inizializzati prima del loro utilizzo.

Esercizio 1 (2) Create una semplice classe; all'interno di una seconda classe definite un riferimento a un oggetto della prima. Utilizzate la tecnica di *lazy initialization* per istanziare questo oggetto.

Sintassi dell'ereditarietà

L'ereditarietà è parte integrante di Java e di tutti i linguaggi orientati agli oggetti, e in un modo o nell'altro viene sempre chiamata in causa: infatti, anche non ereditando esplicitamente da qualche classe, ogni volta che create una classe ereditate implicitamente da quella radice standard di Java, **Object**.

La sintassi di composizione è banale, tuttavia l'ereditarietà ne richiede una speciale. Quando utilizzate l'ereditarietà è come se affermaste che la nuova classe assomiglia a quella precedente: nel codice ribadite questa affermazione prima della parentesi graffa di apertura del corpo della classe, mediante la parola chiave **extends** seguita dal nome della classe di base. Così facendo ottenete in modo automatico tutti i campi e i metodi presenti nella classe di base, come nell'esempio seguente:

```
//: reusing/Detergent.java
// Sintassi e proprietà dell'ereditarietà.
import static net.mindview.util.Print.*;

class Cleanser {
    private String s = "Cleanser";
    public void append(String a) { s += a; }
    public void dilute() { append(" dilute()"); }
    public void apply() { append(" apply()"); }
    public void scrub() { append(" scrub()"); }
    public String toString() { return s; }
    public static void main(String[] args) {
        Cleanser x = new Cleanser();
        x.dilute(); x.apply(); x.scrub();
        print(x);
    }
}
```

```
public class Detergent extends Cleanser {
    // Cambia un metodo:
    public void scrub() {
        append(" Detergent.scrub()");
        super.scrub(); // Chiama la versione della superclasse
    }
    // Aggiunge metodi all'interfaccia:
    public void foam() { append(" foam()"); }
    // Test della nuova classe:
    public static void main(String[] args) {
        Detergent x = new Detergent();
        x.dilute();
        x.apply();
        x.scrub();
        x.foam();
        print(x);
        print("Testing base class:");
        Cleanser.main(args);
    }
} /* Output:
Cleanser dilute() apply() Detergent.scrub() scrub() foam()
Testing base class:
Cleanser dilute() apply() scrub()
*///:~
```

Questo codice evidenzia alcune caratteristiche: per prima cosa, nel metodo **append()** di **Cleanser** le **String** vengono concatenate a **s** mediante l'operatore **+=**, che con **“+”** è uno degli operatori che i progettisti Java hanno sovraccaricato per gestire le stringhe.

In secondo luogo, **Cleanser** e **Detergent** contengono entrambi un metodo **main()**. Potete creare un **main()** per ogni classe, in modo da sottoporle facilmente a verifiche; non occorre che rimuoviate il **main()** quando i test sono terminati, ma potete lasciarlo per una verifica successiva.

Anche se il vostro programma è strutturato in molte classi Java eseguirà soltanto il **main()** di quella invocata dalla riga di comando: in questo caso specifico, digitando **java Detergent** chiamate **Detergent.main()**. Nulla vieta, tuttavia, di digitare **java Cleanser** per richiamare **Cleanser.main()**, benché

Cleanser non sia una classe di tipo **public**. Anche se una classe è definita da un livello di accesso di tipo package, esiste comunque un metodo **main()** **public** accessibile.

Nell'esempio potete vedere che **Detergent.main()** chiama **Cleanser.main()** in modo esplicito passandogli gli stessi argomenti dalla riga di comando: tenete presente, però, che è possibile passare qualunque array **String**.

È importante che tutti i metodi in **Cleanser** siano pubblici: in mancanza di specifici modificatori di accesso, quello predefinito del membro diventerà l'accesso di tipo package, che ammette soltanto i membri del package stesso. Di conseguenza nell'ambito di questo package chiunque potrebbe utilizzare tali metodi: **Detergent**, per esempio, non avrebbe problemi. Tuttavia, qualora una classe di un altro package ereditasse da **Cleanser** potrebbe accedere soltanto ai membri **public**. Come regola generale, pertanto, nell'ottica dell'ereditarietà dovreste rendere tutti i campi **private** e tutti i metodi **public**; anche i membri di tipo **protected** consentono l'accesso da parte di classi derivate, come vedrete meglio nel prosieguo del volume. Naturalmente in casi particolari dovrete attuare adattamenti, tuttavia questa indicazione vi sarà utile.

L'interfaccia di **Cleanser** possiede un insieme di metodi: **append()**, **dilute()**, **apply()**, **scrub()** e **toString()**. Poiché **Detergent** deriva da **Cleanser**, per mezzo della parola chiave **extends**, automaticamente ottiene questi metodi nella propria interfaccia, anche se non tutti sono definiti esplicitamente in **Detergent**. Potete pensare all'ereditarietà, quindi, in termini di riutilizzo di una classe.

Come mostrato nel metodo **scrub()**, è possibile modificare un metodo che è stato definito nella classe di base, per esempio per chiamare il metodo della superclasse dall'interno della nuova versione. All'interno di **scrub()**, però, non potete chiamare semplicemente **scrub()** poiché così facendo produrreste una chiamata ricorsiva, un effetto certamente indesiderabile. Per risolvere il problema Java mette a disposizione la parola chiave **super**, che fa riferimento alla superclasse da cui eredita la classe corrente: l'espressione **super.scrub()** chiama la versione del metodo **scrub()** appartente alla classe di base.

L'ereditarietà non vi limita a utilizzare i soli metodi della classe di base ma consente di integrare la classe derivata con nuovi metodi, allo stesso modo in cui aggiungereste un metodo a una classe: definendolo. Il metodo **foam()** è un esempio di questa tecnica.

Detergent.main() mostra che per un oggetto **Detergent** è possibile chiamare tutti i metodi che sono disponibili sia in **Cleanser** sia in **Detergent**, per esempio **foam()**.

Esercizio 2 (2) Ereditate una nuova classe dalla classe **Detergent**. Sovrascrivete **scrub()** e aggiungete un nuovo metodo chiamato **sterilize()**.

Inizializzazione della superclasse

Poiché ora sono coinvolte due classi, la superclasse e quella derivata, potrebbe essere difficile concepire l'oggetto prodotto da una classe derivata. Dall'esterno è come se la nuova classe avesse la stessa interfaccia di quella di base, con l'aggiunta di eventuali altri metodi e campi. L'ereditarietà non si limita tuttavia a copiare l'interfaccia della classe di base. Al momento della sua creazione, un oggetto della classe derivata contiene un "sotto-oggetto" (*subobject*) della superclasse, lo stesso che vi ritrovereste creando un oggetto della classe di base: l'unica differenza è che, visto dall'esterno, il sotto-oggetto della classe di base è "avvolto" nell'oggetto della classe derivata.

È essenziale che il sotto-oggetto della superclasse sia inizializzato in modo corretto, e c'è un solo modo per garantirlo: eseguire l'inizializzazione nel costruttore chiamando quello della superclasse, che ha tutte le informazioni e i privilegi necessari per procedere all'inizializzazione della classe di base. A questo scopo, Java include automaticamente chiamate al costruttore della superclasse nel costruttore della classe derivata. Quello che segue è un esempio che illustra questo meccanismo con tre livelli di ereditarietà.

```
//: reusing/Cartoon.java
// Chiamate al costruttore durante l'ereditarieta'.
import static net.mindview.util.Print.*;

class Art {
    Art() { print("Art constructor"); }
}

class Drawing extends Art {
    Drawing() { print("Drawing constructor"); }
}

public class Cartoon extends Drawing {
    public Cartoon() { print("Cartoon constructor"); }
    public static void main(String[] args) {
        Cartoon x = new Cartoon();
    }
} /* Output:
Art constructor
```



```
Drawing constructor
Cartoon constructor
*///:~
```

Come potete osservare la costruzione inizia dal centro (la classe base) verso l'esterno, affinché la superclasse sia inizializzata prima che i costruttori delle classi derivate possano accedervi. Anche se non create un costruttore per **Cartoon()**, il compilatore genererà un costruttore predefinito che chiama quello della superclasse.

Esercizio 3 (2) Dimostrate la veridicità dell'affermazione precedente per quanto riguarda il comportamento del compilatore.

Esercizio 4 (2) Dimostrate che i costruttori delle classi di base vengono a) sempre chiamati, e b) sempre chiamati prima dei costruttori delle classi derivate.

Esercizio 5 (1) Create due classi, **A** e **B**, con costruttori predefiniti (elenchi di argomenti vuoti) contenenti un'istruzione che ne dimostri la presenza. Create una nuova classe **C** che eredita da **A**, e generate un membro della classe **B** all'interno di **C**. Non create un costruttore per **C**; generate un oggetto della classe **C** e osservate i risultati.

Costruttori con argomenti

L'esempio precedente ha costruttori predefiniti, ossia privi di argomenti, facilmente richiamabili dal compilatore poiché non danno luogo a incertezza sul tipo di argomenti da passare. Qualora la superclasse non abbia un costruttore predefinito o se volete chiamare un costruttore di superclasse con argomenti, dovrete scrivere appositamente una chiamata a questo costruttore utilizzando la parola chiave **super** e l'elenco di argomenti appropriato.

```
//: reusing/Chess.java
// Ereditarieta', costruttori e argomenti.
import static net.mindview.util.Print.*;

class Game {
    Game(int i) {
        print("Game constructor");
    }
}
```



```
class BoardGame extends Game {
    BoardGame(int i) {
        super(i);
        print("BoardGame constructor");
    }
}

public class Chess extends BoardGame {
    Chess() {
        super(11);
        print("Chess constructor");
    }
    public static void main(String[] args) {
        Chess x = new Chess();
    }
} /* Output:
Game constructor
BoardGame constructor
Chess constructor
*///:~
```

Se non chiamate il costruttore della superclasse in **BoardGame()**, per esempio, il compilatore segnalerà che non riesce a trovare un costruttore della classe **Game()**. Inoltre la chiamata al costruttore della classe di base deve essere la prima operazione prevista nel costruttore della classe derivata: se così non fosse, il compilatore ve lo segnalerà.

Esercizio 6 (1) Utilizzando **Chess.java** verificate le istruzioni del paragrafo precedente.

Esercizio 7 (1) Modificate l'Esercizio 5 in modo che **A** e **B** abbiano costruttori con argomenti invece di quelli predefiniti. Scrivete un costruttore per **C** ed eseguite tutte le operazioni di inizializzazione al suo interno.

Esercizio 8 (1) Create una classe di base con un solo costruttore non predefinito e una classe derivata che disponga sia di un costruttore predefinito (senza argomenti) sia di un costruttore non predefinito. Nei costruttori della classe derivata chiamate quello della superclasse.



Esercizio 9 (2) Create una classe chiamata **Root** contenente un’istanza per ciascuna delle classi **Component1**, **Component2** e **Component3**, anch’esse da creare. Generate una classe **Stem** che eredita da **Root**, e che contiene un’istanza di ogni “componente”. Tutte le classi dovranno avere costruttori predefiniti che visualizzano un messaggio informativo sulla classe.

Esercizio 10 (1) Modificate l’esercizio precedente in modo che ogni classe abbia soltanto costruttori non predefiniti.

Delega

Un terzo tipo di relazione, non direttamente supportato da Java, è la cosiddetta delega. Essa si colloca a metà strada tra l’ereditarietà e la composizione poiché, come la prima, prevede che la classe da costruire includa un oggetto membro, e nello stesso tempo espone tutti i metodi dell’oggetto membro nella nuova classe, come l’ereditarietà. Considerate per esempio un’astronave, che necessita di un modulo di controllo.

```
//: reusing/SpaceShipControls.java

public class SpaceShipControls {
    void up(int velocity) {}
    void down(int velocity) {}
    void left(int velocity) {}
    void right(int velocity) {}
    void forward(int velocity) {}
    void back(int velocity) {}
    void turboBoost() {}
} ///:~
```

Un modo per costruire un’astronave è l’utilizzo dell’ereditarietà.

```
//: reusing/SpaceShip.java

public class SpaceShip extends SpaceShipControls {
    private String name;
    public SpaceShip(String name) { this.name = name; }
    public String toString() { return name; }
    public static void main(String[] args) {
```

```
SpaceShip protector = new SpaceShip("NSEA Protector");
protector.forward(100);
}
} ///:~
```

Un’astronave (**SpaceShip**), però, non è esattamente “un tipo di” **SpaceShip-Controls** benché si possa “impartirle degli ordini”: per avanzare, per esempio, con **forward()**. È di certo più preciso affermare che la **SpaceShip** contiene gli **SpaceShipControls**, e nello stesso tempo che tutti i metodi in **SpaceShipControls** sono “esposti” (messi a disposizione) nella **SpaceShip**. La soluzione consiste nel ricorrere alla delega.

//: reusing/SpaceShipDelegation.java

```
public class SpaceShipDelegation {
    private String name;
    private SpaceShipControls controls =
        new SpaceShipControls();
    public SpaceShipDelegation(String name) {
        this.name = name;
    }
    // Metodi delegati:
    public void back(int velocity) {
        controls.back(velocity);
    }
    public void down(int velocity) {
        controls.down(velocity);
    }
    public void forward(int velocity) {
        controls.forward(velocity);
    }
    public void left(int velocity) {
        controls.left(velocity);
    }
    public void right(int velocity) {
        controls.right(velocity);
    }
}
```



```

public void turboBoost() {
    controls.turboBoost();
}
public void up(int velocity) {
    controls.up(velocity);
}
public static void main(String[] args) {
    SpaceShipDelegation protector =
        new SpaceShipDelegation("NSEA Protector");
    protector.forward(100);
}
} //:~

```

In questo codice potete vedere che i metodi sono trasferiti all'oggetto **controls** sottostante, di conseguenza l'interfaccia è la stessa di quella che si avrebbe con l'ereditarietà. Tuttavia la delega offre un maggiore controllo, poiché consente di scegliere di fornire soltanto un sottoinsieme dei metodi presenti nell'oggetto membro.

Sebbene il linguaggio Java non supporti la delega, spesso gli strumenti di sviluppo lo fanno: l'esempio precedente, in particolare, è stato generato automaticamente mediante il software IDE Idea di JetBrains.

Esercizio 11 (3) Modificate **Detergent.java** in modo che utilizzi la delega.

Combinazione di composizione ed ereditarietà

L'impiego combinato di composizione ed ereditarietà è una tecnica piuttosto comune. Il seguente esempio illustra la creazione di una classe più complessa che utilizza entrambe le tecniche, oltre alla necessaria inizializzazione del costruttore.

```

//: reusing/PlaceSetting.java
// Combinazione di composizione ed ereditarieta'.
import static net.mindview.util.Print.*;
public class PlaceSetting {
    class Plate {
        Plate(int i) {
            print("Plate constructor");
        }
    }
    class Utensil {
        Utensil(int i) {
            print("Utensil constructor");
        }
    }
    class Spoon extends Utensil {
        Spoon(int i) {
            super(i);
            print("Spoon constructor");
        }
    }
    class Fork extends Utensil {
        Fork(int i) {
            super(i);
            print("Fork constructor");
        }
    }
    class Knife extends Utensil {
        Knife(int i) {
            super(i);
            print("Knife constructor");
        }
    }
}

```



```

        print("Plate constructor");
    }
}
}

class DinnerPlate extends Plate {
    DinnerPlate(int i) {
        super(i);
        print("DinnerPlate constructor");
    }
}

class Utensil {
    Utensil(int i) {
        print("Utensil constructor");
    }
}

class Spoon extends Utensil {
    Spoon(int i) {
        super(i);
        print("Spoon constructor");
    }
}

class Fork extends Utensil {
    Fork(int i) {
        super(i);
        print("Fork constructor");
    }
}

class Knife extends Utensil {
    Knife(int i) {
        super(i);
        print("Knife constructor");
    }
}

```



```
// Un modo "artistico" di procedere:
class Custom {
    Custom(int i) {
        print("Custom constructor");
    }
}

public class PlaceSetting extends Custom {
    private Spoon sp;
    private Fork frk;
    private Knife kn;
    private DinnerPlate pl;
    public PlaceSetting(int i) {
        super(i + 1);
        sp = new Spoon(i + 2);
        frk = new Fork(i + 3);
        kn = new Knife(i + 4);
        pl = new DinnerPlate(i + 5);
        print("PlaceSetting constructor");
    }
    public static void main(String[] args) {
        PlaceSetting x = new PlaceSetting(9);
    }
} /* Output:
Custom constructor
Utensil constructor
Spoon constructor
Utensil constructor
Fork constructor
Utensil constructor
Knife constructor
Plate constructor
DinnerPlate constructor
PlaceSetting constructor
*///:~
```



Nonostante il compilatore costringa a inizializzare le classi di base e richieda che lo facciate subito all'inizio del costruttore, non controlla che inizializziate gli oggetti membro, pertanto dovete prestare attenzione a questo dettaglio. Osservate il modo sorprendentemente pulito con cui le classi sono mantenute separate; per riutilizzare il codice non occorre neppure il codice sorgente dei metodi, al massimo dovrete importare un package: questo, peraltro, è vero sia per l'ereditarietà sia per la composizione.

Come garantire un cleanup corretto

Avete visto che Java non ha il concetto di distruttore, tipico di C++, vale a dire un metodo chiamato automaticamente quando un oggetto viene distrutto. Probabilmente il motivo è che in Java si è soliti "dimenticare" semplicemente gli oggetti invece che distruggerli, lasciando al garbage collector il compito di recuperare la memoria necessaria.

Tale tecnica è spesso sufficiente, tuttavia in alcune situazioni durante la vita della vostra classe potreste eseguire attività che richiedono un successivo cleanup. Come si è detto nel Capitolo 5, non è possibile sapere né se il garbage collector verrà chiamato né se lo sarà: per procedere a una sorta di cleanup in una classe, quindi, dovete scrivere un metodo speciale e assicurarvi che il programmatore client sappia di poterlo chiamare. Oltre a questo, come vedrete nel Volume 2, Capitolo 1, dedicato alla gestione degli errori, dovete cauterlarvi da un'eventuale eccezione da parte del compilatore, inserendo il metodo di cleanup in una clausola **finally**.

Considerate l'esempio di un sistema di progettazione assistita (CAD) che disegna immagini sullo schermo.

```
//: reusing/CADSystem.java
// Assicura il cleanup corretto.
package reusing;
import static net.mindview.util.Print.*;

class Shape {
    Shape(int i) { print("Shape constructor"); }
    void dispose() { print("Shape dispose "); }
}

class Circle extends Shape {
    Circle(int i) {
```



256

```
super(i);
print("Drawing Circle");
}
void dispose() {
print("Erasing Circle");
super.dispose();
}
}

class Triangle extends Shape {
Triangle(int i) {
super(i);
print("Drawing Triangle");
}
void dispose() {
print("Erasing Triangle");
super.dispose();
}
}

class Line extends Shape {
private int start, end;
Line(int start, int end) {
super(start);
this.start = start;
this.end = end;
print("Drawing Line: " + start + ", " + end);
}
void dispose() {
print("Erasing Line: " + start + ", " + end);
super.dispose();
}
}

public class CADSystem extends Shape {
private Circle c;
```



257

```
private Triangle t;
private Line[] lines = new Line[3];
public CADSystem(int i) {
super(i + 1);
for(int j = 0; j < lines.length; j++)
lines[j] = new Line(j, j*j);
c = new Circle(1);
t = new Triangle(1);
print("Combined constructor");
}
public void dispose() {
print("CADSystem.dispose()");
// L'ordine di cleanup è inverso rispetto
// all'ordine di inizializzazione:
t.dispose();
c.dispose();
for(int i = lines.length - 1; i >= 0; i--)
lines[i].dispose();
super.dispose();
}
public static void main(String[] args) {
CADSystem x = new CADSystem(47);
try {
// Gestione del codice e delle eccezioni...
} finally {
x.dispose();
}
}
} /* Output:
Shape constructor
Shape constructor
Drawing Line: 0, 0
Shape constructor
Drawing Line: 1, 1
Shape constructor
Drawing Line: 2, 4
```



```

Shape constructor
Drawing Circle
Shape constructor
Drawing Triangle
Combined constructor
CADSystem.dispose()
Erasing Triangle
Shape dispose
Erasing circle
Shape dispose
Erasing Line: 2, 4
Shape dispose
Erasing Line: 1, 1
Shape dispose
Erasing Line: 0, 0
Shape dispose
Shape dispose
*///:~

```

In questa applicazione tutto viene rappresentato come oggetto **Shape**, che a sua volta è anch'esso un tipo di **Object** poiché eredita implicitamente dalla classe radice: ogni classe sovrscrive il metodo **dispose()** di **Shape**, oltre a chiamare la versione superclasse di questo metodo, tramite **super**.

Tutte le classi specifiche di **Shape** (**Circle**, **Triangle** e **Line**) hanno costruttori che “disegnano”; tenete presente, in ogni caso, che qualsiasi altro metodo chiamato durante la vita dell'oggetto potrebbe eseguire qualcosa che richiede un successivo cleanup.

Ogni classe ha il proprio metodo **dispose()** che ripristina tutto ciò che non fa capo alla memoria, ristabilendo lo stato precedente all'esistenza dell'oggetto.

In **main()** vi sono due nuove parole chiave che esaminerete nel Volume 2, Capitolo 1, dedicato alla gestione degli errori: **try** e **finally**. La parola chiave **try** indica che il blocco di codice successivo, delimitato da parentesi graffe, è un'area sorvegliata che deve essere tenuta sotto osservazione: uno dei possibili “trattamenti speciali” cui può essere soggetta è che il codice presente nella clausola **finally** successiva a quest'area venga sempre eseguito, a prescindere da come avviene l'uscita dal blocco **try**. Come scoprirete a proposito della gestione delle eccezioni, infatti, è possibile uscire da un blocco **try** in



diversi modi: in questo caso specifico la clausola **finally** indica di chiamare il metodo **dispose()** per **x**, qualsiasi cosa accada.

Nel vostro metodo di cleanup, in questo caso **dispose()**, dovete anche prestare attenzione all'ordine di chiamata dei metodi di cleanup della superclasse e dell'oggetto membro, qualora un sotto-oggetto dipenda da un altro. Di norma dovreste adottare lo stesso criterio imposto da un compilatore C++ ai propri distruttori: eseguire innanzitutto l'attività di cleanup specifica per le vostre classi, in ordine inverso di creazione; in generale, questo richiede che gli elementi della classe di base siano ancora istanziabili. Soltanto in seguito chiamerete il metodo di cleanup della classe di base, come illustrato in questo esempio.

Nella maggior parte dei casi l'attività di cleanup non presenta problemi, ed è sufficiente permettere al garbage collector di svolgere il suo lavoro; tuttavia l'esecuzione esplicita del cleanup richiede estrema attenzione e impegno, poiché il garbage collector non offre molte garanzie: potrebbe non essere mai chiamato a intervenire e, quando lo sia, richiamare gli oggetti nell'ordine che ritiene opportuno. Tutto ciò su cui potete contare è il recupero della memoria inutilizzata. Se volete che il cleanup abbia luogo, evitate di servirvi di **finalize()** e scrivete voi stessi i metodi che occorrono.

Esercizio 12 (3) Aggiungete una gerarchia corretta di metodi **dispose()** a tutte le classi dell'Esercizio 9.

Occultamento del nome

A differenza di quanto avviene in C++, se in una superclasse Java il nome di un metodo viene sovraccaricato più volte, la sua ridefinizione nella classe non nasconderà nessuna delle versioni della classe di base. Di conseguenza l'overload funziona indipendentemente dal fatto che il metodo sia stato definito al livello corrente o in una classe di base.

```

//: reusing/Hide.java
// L'overload di un nome di metodo di una superclasse in una
// classe derivata
// non occulta le versioni della superclasse.
import static net.mindview.util.Print.*;

class Homer {
    char doh(char c) {
        print("doh(char)");
    }
}

```



```

        return 'd';
    }

    float doh(float f) {
        print("doh(float)");
        return 1.0f;
    }

}

class Milhouse {}

class Bart extends Homer {
    void doh(Milhouse m) {
        print("doh(Milhouse)");
    }
}

public class Hide {
    public static void main(String[] args) {
        Bart b = new Bart();
        b.doh(1);
        b.doh('x');
        b.doh(1.0f);
        b.doh(new Milhouse());
    }
} /* Output:
doh(float)
doh(char)
doh(float)
doh(Milhouse)
*///:~

```

Potete notare che tutti i metodi sovraccarichi di **Homer** sono disponibili in **Bart**, sebbene **Bart** introduca un nuovo metodo sovraccarico; in C++ questa condizione nasconderebbe i metodi della classe di base. Come vedrete nel prossimo capitolo, è prassi comune sovrascrivere i metodi omonimi utilizzando l'esatta segnatura e lo stesso tipo di ritorno della superclasse. Un diverso comportamento potrebbe generare confusione, e questo spiega il

motivo per cui C++ non ammette il ricorso a questa tecnica: per evitarvi di eseguire qualcosa che si rivelerebbe probabilmente un errore.

Java SE5 ha reso disponibile l'annotazione **@Override**, che pur non essendo una parola chiave può essere utilizzata come se lo fosse. Quando ritenete opportuno sovrascrivere un metodo, potete aggiungere questa annotazione: in questo modo, se eseguite inavvertitamente un sovraccarico (*overload*) invece di una sovrascrittura (*override*), il compilatore visualizzerà un messaggio di errore.

```

//: reusing/Lisa.java
// {CompileTimeError} (Non compilera')

class Lisa extends Homer {
    @Override void doh(Milhouse m) {
        System.out.println("doh(Milhouse)");
    }
} ///:~

```

Il tag **{CompileTimeError}** esclude il file dal build Ant di questo manuale, tuttavia se compilate il codice direttamente otterrete il messaggio seguente:

method does not override a method from its superclass

L'annotazione **@Override**, come vedete, impedisce di produrre un overload non espressamente voluto.

Esercizio 13 (2) Create una classe con un metodo che viene sovraccaricato per tre volte. Da questa ereditate una nuova classe, aggiungete un nuovo overload del metodo e dimostrate che i quattro metodi sono disponibili nella classe derivata.

Come scegliere tra composizione ed ereditarietà

La composizione e l'ereditarietà permettono entrambe di includere sotto-oggetti nella vostra nuova classe: la composizione lo fa in modo esplicito mentre nell'ereditarietà il comportamento è implicito. Potrete quindi chiedervi quali sono le differenze tra le due tecniche e quando optare per l'una o l'altra.

La composizione è utilizzata generalmente per usufruire delle funzionalità di una classe esistente all'interno della nuova classe, senza per questo dispor-



re della sua interfaccia. In pratica l'oggetto viene incorporato in modo da essere utilizzabile per l'implementazione di funzionalità nella nuova classe, tuttavia il programmatore di quest'ultima lavorerà con l'interfaccia definita per la nuova classe anziché con quella dell'oggetto incorporato. Questo comportamento si ottiene incorporando nella nuova classe gli oggetti privati (**private**) delle classi esistenti.

Talvolta può essere opportuno permettere all'utente della classe di accedere direttamente alla composizione di quella nuova, ossia rendere pubblici (**public**) gli oggetti membro; questi oggetti beneficiano anch'essi dell'occultamento dell'implementazione, pertanto la tecnica è perfettamente sicura. Il fatto che il programmatore client sappia che avete assemblato un certo numero di componenti rende l'interfaccia più facile da capire. Considerate un esempio tra i più classici, l'automobile (**car**).

```
//: reusing/Car.java
// Composizione con oggetti di tipo public.

class Engine {
    public void start() {}
    public void rev() {}
    public void stop() {}
}

class Wheel {
    public void inflate(int psi) {}
}

class Window {
    public void rollup() {}
    public void rolldown() {}
}

class Door {
    public Window window = new Window();
    public void open() {}
    public void close() {}
}
```



```
public class Car {
    public Engine engine = new Engine();
    public Wheel[] wheel = new Wheel[4];
    public Door
        left = new Door(),
        right = new Door(); // A due porte
    public Car() {
        for(int i = 0; i < 4; i++)
            wheel[i] = new Wheel();
    }
    public static void main(String[] args) {
        Car car = new Car();
        car.left.window.rollup();
        car.wheel[0].inflate(72);
    }
} ///:~
```

Visto che in questo caso l'assemblaggio di un'automobile non si limita a considerare la progettazione, bensì è parte integrante dell'analisi del problema, il fatto di rendere **public** i membri facilita la comprensione dell'utilizzo della classe (da parte del programmatore client) e richiede codice meno complesso (da parte dello sviluppatore della classe). Ricordate comunque che questo è un caso speciale: normalmente dovreste rendere i campi **private**.

Quando implementate l'ereditarietà prendete una classe esistente e ne realizzate una versione speciale. Di solito questo equivale a rendere specializzata una certa classe generica, per adattarla a una particolare necessità.

Riflettendoci attentamente vi renderete conto che non avrebbe senso comporre un'automobile utilizzando un oggetto veicolo, poiché l'automobile non contiene un veicolo, ma è essa stessa un veicolo. La relazione *è-un* (*is-a*) viene manifestata dall'ereditarietà, mentre *ha-un* (*has-a*) è espressa dalla composizione.

Esercizio 14 (1) In **Car.java** aggiungete il metodo **service()** a **Engine** e chiamate questo metodo in **main()**.



L'accesso protected

Ora che avete visto come funziona l'ereditarietà, la parola chiave **protected** assume finalmente una sua coerenza. In un mondo ideale la parola chiave **private** sarebbe sufficiente, ma nei progetti reali talvolta occorre nascondere qualcosa e nello stesso tempo permettere l'accesso ai membri delle classi derivate.

In quest'ottica la parola chiave **protected** è la sintesi del pragmatismo: in pratica equivale ad affermare che un determinato oggetto è privato per l'utente della classe, ma disponibile a qualsiasi classe eredita da quella corrente o a qualsiasi altra appartenente allo stesso package; tenete presente che **protected** garantisce anche l'accesso al package.

Malgrado sia possibile creare campi **protected**, l'approccio migliore è lasciare i campi **private**, poiché dovreste sempre conservarvi il diritto di modificare l'implementazione sottostante; usando metodi **protected** potrete poi permettere l'accesso controllato alle classi che ereditano dalla vostra.

```
//: reusing/Orc.java
// La parola chiave protected.
import static net.mindview.util.Print.*;

class Villain {
    private String name;
    protected void set(String nm) { name = nm; }
    public Villain(String name) { this.name = name; }
    public String toString() {
        return "I'm a Villain and my name is" + name;
    }
}

public class Orc extends Villain {
    private int orcNumber;
    public Orc(String name, int orcNumber) {
        super(name);
        this.orcNumber = orcNumber;
    }
    public void change(String name, int orcNumber) {
        set(name); // Disponibile perche' protected
    }
}
```

```
this.orcNumber = orcNumber;
}
public String toString() {
    return "Orc " + orcNumber + ":" + super.toString();
}
public static void main(String[] args) {
    Orc orc = new Orc("Limburger", 12);
    print(orc);
    orc.change("Bob", 19);
    print(orc);
}
} /* Output:
Orc 12: I'm a Villain and my name is Limburger
Orc 19: I'm a Villain and my name is Bob
*///:~
```

Noteate che **change()** può accedere a **set()** poiché è **protected**, e osservate come il metodo **toString()** di **Orc** sia definito in termini di versione superclasse di **toString()**.

Esercizio 15 (2) All'interno di un package create una classe contenente un metodo **protected**; dall'esterno del package provate a chiamare il metodo protetto e motivate i risultati. Quindi ereditate da questa classe, poi, dall'interno di un metodo della nuova classe derivata, chiamate il metodo **protected**.

Upcasting

L'aspetto più importante dell'ereditarietà non è tanto il fatto di fornire metodi alla nuova classe, quanto la relazione espressa tra la nuova classe e la superclasse, una relazione che può essere riassunta affermando che la nuova classe è un tipo della classe esistente.

Questa descrizione non è soltanto un modo atipico per illustrare l'ereditarietà ma è supportata direttamente dal linguaggio. Considerate, per esempio, una classe di base chiamata **Instrument** che rappresenta gli strumenti musicali, e una derivata chiamata **Wind** (ottoni). Poiché l'ereditarietà garantisce che tutti i metodi della classe di base sono disponibili anche nella classe derivata, qualsiasi messaggio che può essere inviato alla superclasse può essere trasmesso anche alla classe derivata. Se la classe **Instrument**

possiede un metodo **play()**, lo stesso metodo sarà disponibile in **Wind**: ciò equivale ad affermare con precisione che un oggetto **Wind** è anche un tipo di **Instrument**.

L'esempio seguente illustra il modo in cui il compilatore implementa questo concetto.

```
//: reusing/Wind.java
// Ereditarieta' e upcasting.

class Instrument {
    public void play() {}
    static void tune(Instrument i) {
        // ...
        i.play();
    }
}

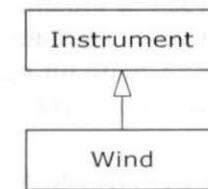
// Gli oggetti Wind sono strumenti
// perche' hanno la stessa interfaccia:
public class Wind extends Instrument {
    public static void main(String[] args) {
        Wind flute = new Wind();
        Instrument.tune(flute); // Upcasting
    }
} ///:~
```

Ciò che è interessante in questo codice è il metodo **tune()**, che accetta un riferimento **Instrument**: notate, tuttavia, che in **Wind.main()** al metodo **tune()** viene passato un riferimento a **Wind**. Tenuto conto che Java è così esclusivo nel controllo dei tipi, potrebbe apparirvi strano che un metodo che accetta un tipo sia così disponibile ad accettarne un altro: tutto apparirà chiaro non appena vi renderete conto che un oggetto **Wind** è anche un oggetto **Instrument** e che non esiste metodo di **Instrument** richiamabile da **tune()** che non lo sia anche in **Wind**.

All'interno di **tune()** il codice funziona per **Instrument** e per tutto ciò che deriva da questa classe, e l'azione di convertire un riferimento **Wind** in un riferimento **Instrument** è chiamata *upcasting*.

Perché “*upcasting*”?

Il termine deriva dal modo in cui tradizionalmente si disegnano i diagrammi di ereditarietà delle classi, con la radice in alto che si dirama verso il basso; nulla vi vieta, ovviamente, di adottare il metodo che preferite. Il diagramma di ereditarietà per **Wind.java** è mostrato nella figura seguente.



Nel diagramma di ereditarietà, il casting da un tipo derivato in un tipo di base procede verso l'alto, ed è quindi comunemente definito *upcasting*. L'operazione di *upcasting* è sempre considerata sicura poiché procede da un tipo più specifico a uno più generale: in pratica la classe derivata è un sovrainsieme della classe di base, che potrebbe contenere più metodi rispetto alla superclasse, ma in ogni caso deve possedere almeno quelli della classe di base. L'unico inconveniente che può verificarsi nell'interfaccia di classe durante l'*upcast* è la perdita di metodi, che potrebbero non essere conseguiti. Questo è il motivo per cui il compilatore ammette l'*upcasting* senza alcun cast esplicito né altra notazione speciale.

È possibile eseguire anche l'operazione inversa, il cosiddetto *downcasting*, tuttavia questo implica un problema che esaminerete nel Capitolo 8 e nel Volume 2, Capitolo 2 dedicato alle informazioni sui tipi.

Nuovo confronto tra composizione ed ereditarietà

Nella programmazione a oggetti la tecnica più ricorrente per creare e utilizzare il codice è il semplice assemblaggio di dati e metodi nella stessa classe, e l'utilizzo di oggetti di quella classe. Utilizzerete anche classi esistenti per costruirne di nuove, ricorrendo alla composizione, e con minore frequenza vi servirete dell'ereditarietà. Quindi, il fatto che l'ereditarietà sia così ampiamente enfatizzata nello studio della programmazione OOP non significa doverla utilizzare non appena possibile: al contrario, dovreste ricorrervi con parsimonia e soltanto quando ne sia evidente l'utilità. Uno dei modi più pratici per determinare quando utilizzare la composizione o l'ereditarietà è chiedersi se possa risultare necessario eseguire un *upcast* dalla nuova classe a quella di base.



Qualora dobbiate eseguire l'upcast è evidente che l'ereditarietà è necessaria, in caso contrario dovreste pensare alla possibilità di non utilizzarla.

Nel Capitolo 8 vedrete una delle ragioni più irresistibili per procedere a un upcasting, ma se ricorderete di chiedervi "Ho davvero bisogno di un upcast?", sarete in grado di valutare con ragionevole precisione quando scegliere la composizione o l'ereditarietà.

Esercizio 16 (2) Create una classe chiamata **Amphibian**, da cui per ereditarietà genererete una classe chiamata **Frog**; inserite metodi appropriati nella classe di base. Create un oggetto **Frog** ed eseguitene l'upcast ad **Amphibian**, dimostrando che tutti i metodi funzionano ancora.

Esercizio 17 (1) Modificate l'Esercizio 16 in modo che **Frog** contenga le versioni sovrascritte delle definizioni di metodo della classe di base, ossia fornite nuove definizioni che utilizzano le stesse segnature di metodo. Prendete nota del comportamento di **main()**.

La parola chiave final

In Java la parola chiave **final** assume significati lievemente diversi in funzione del contesto, tuttavia di norma indica che un oggetto non può essere modificato. Potreste decidere di impedire le modifiche per ragioni di progettazione o di efficienza. Dal momento che questi due motivi sono alquanto diversi, è possibile che facciate un uso improprio della parola chiave **final**.

Nei prossimi paragrafi esaminerete in dettaglio le tre situazioni in cui può essere utilizzata **final**: per i dati, i metodi e le classi.

Dati di tipo final

Molti linguaggi di programmazione permettono di segnalare al compilatore che un certo dato è "costante". Una costante è utile perché:

1. può essere una costante di compilazione che non cambierà mai;
2. può essere un valore inizializzato in fase d'esecuzione che non volete venga modificato.

Nel caso di una costante di compilazione è permesso "incorporare" al compilatore il valore costante in qualsiasi calcolo venga utilizzato; questo significa che il calcolo può essere eseguito al momento della compilazione, riducendo l'onere elaborativo in fase di esecuzione. In Java le costanti di questo genere devono essere tipi primitivi e vengono espressi dalla parola



chiave **final**; al momento della definizione, a questo tipo di costante deve essere assegnato un valore.

Un campo che sia simultaneamente **static** e **final** ha soltanto una locazione di memoria, che non è modificabile.

Quando utilizzate **final** con riferimenti di oggetto anziché con tipi primitivi, il risultato può essere fuorviante: in caso di primitivo, **final** rende costante il valore, mentre con un riferimento di oggetto **final** rende costante il riferimento. Una volta che il riferimento è stato inizializzato a un oggetto non può essere variato per puntare a un altro. L'oggetto stesso, tuttavia, rimane modificabile poiché Java non ha modo di trasformare in costante un oggetto vero e proprio; naturalmente potete sempre realizzare la classe in modo che gli oggetti abbiano l'effetto di apparire costanti. Le limitazioni descritte si applicano anche agli array, anch'essi considerati oggetti.

L'esempio seguente illustra il funzionamento dei campi **final**; tenete presente che per convenzione tutti i campi che sono sia **static** sia **final**, cioè le costanti di compilazione, sono scritti in maiuscolo e separando le parole mediante caratteri di sottolineatura: è lo stesso accorgimento utilizzato dalle costanti di C, da cui ha avuto origine questa convenzione, ormai pressoché universale.

```
//: reusing/FinalData.java
// L'effetto di final sui campi.
import java.util.*;
import static net.mindview.util.Print.*;

class Value {
    int i; // Accesso al package
    public Value(int i) { this.i = i; }
}

public class FinalData {
    private static Random rand = new Random(47);
    private String id;
    public FinalData(String id) { this.id = id; }
    // Possono essere costanti di compilazione:
    private final int valueOne = 9;
    private static final int VALUE_TWO = 99;
    // Tipica costante public:
```



```

public static final int VALUE_THREE = 39;
// Non possono essere costanti di compilazione:
private final int i4 = rand.nextInt(20);
static final int INT_5 = rand.nextInt(20);
private Value v1 = new Value(11);
private final Value v2 = new Value(22);
private static final Value VAL_3 = new Value(33);
// Array:
private final int[] a = { 1, 2, 3, 4, 5, 6 };
public String toString() {
    return id + ":" + i4 + ", INT_5 = " + INT_5;
}
public static void main(String[] args) {
    FinalData fd1 = new FinalData("fd1");
    //! fd1.valueOne++; /* Errore: non e' possibile cambiare
    //                           il valore*/
    fd1.v2.i++; // Object non e' una costante!
    fd1.v1 = new Value(9); // OK -- non final
    for(int i = 0; i < fd1.a.length; i++)
        fd1.a[i]++; // Object non e' una costante!
    //! fd1.v2 = new Value(0);      // Errore: non e' possibile
    //! fd1.VAL_3 = new Value(1); // cambiare il riferimento
    //! fd1.a = new int[3];
    print(fd1);
    print("Creating new FinalData");
    FinalData fd2 = new FinalData("fd2");
    print(fd1);
    print(fd2);
}
} /* Output:
fd1: i4 = 15, INT_5 = 18
Creating new FinalData
fd1: i4 = 15, INT_5 = 18
fd2: i4 = 13, INT_5 = 18
*///:~
```

Poiché **valueOne** e **VALUE_TWO** sono primitivi **final** con valori di compilazione, essi possono essere impiegati entrambi come costanti di compilazione e non presentano differenze importanti. **VALUE_THREE** è la forma più tipica in cui troverete definite tali costanti: **public** per essere utilizzabili fuori del pacchetto, **static** per evidenziare che ne esiste una soltanto e **final** per indicare che si tratta di una costante.

Naturalmente il solo fatto che un elemento sia **final** non implica che il suo valore sia noto al momento della compilazione, come dimostra l'inizializzazione di **i4** e **INT_5** in fase di esecuzione mediante numeri generati casualmente.

Nella stessa porzione di codice è illustrata anche la differenza tra un valore **final static** e uno **final non static**: il compilatore considera allo stesso modo i valori in fase di compilazione (e probabilmente li ottimizza), pertanto tale differenza si evidenzia solo quando i valori vengono inizializzati in fase di esecuzione e viene visualizzata eseguendo il programma. Notate che i valori di **i4** per **fd1** e **fd2** sono univoci, mentre il valore di **INT_5** non viene modificato dalla creazione del secondo oggetto **FinalData**: questo accade poiché, essendo **static**, viene inizializzato al caricamento e non ogni volta si crea un nuovo oggetto.

Le variabili da **v1** a **VAL_3** dimostrano il significato di un riferimento **final**. Come potete vedere in **main()**, il fatto che **v2** sia **final** non significa che sia impossibile alterarne il valore: trattandosi di un riferimento, **final** indica che non è permesso collegare nuovamente **v2** a un nuovo oggetto. Potete anche notare che lo stesso significato è valido per un array, che non è altro che un diverso tipo di riferimento; tenete presente, inoltre, che all'apparenza Java non permette di rendere **final** i riferimenti presenti negli array. In sostanza, sembra che rendere **final** i riferimenti sia meno utile che rendere **final** i primitivi.

Esercizio 18 (2) Create una classe con un campo **final static** e un campo **final** e dimostrate la differenza tra i due.

Campi final non inizializzati

Java permette la creazione dei cosiddetti *blank final*, ovvero campi dichiarati come **final** ma ai quali non è assegnato un valore di inizializzazione: tali campi devono essere sempre inizializzati prima del loro utilizzo, e il compilatore garantisce che ciò avvenga.

Tuttavia i blank final offrono una maggiore flessibilità nell'utilizzo della parola chiave **final**, dal momento che, per esempio, mediante questa tecnica un



campo **final** di una classe può essere diverso per ogni oggetto, pur mantenendo la sua caratteristica di invariabilità. Considerate l'esempio seguente:

```
//: reusing/BlankFinal.java
// I campi "blank" final.

class Poppet {
    private int i;
    Poppet(int ii) { i = ii; }
}

public class BlankFinal {
    private final int i = 0; // Final inizializzato
    private final int j; // Blank final
    private final Poppet p; // Riferimento blank final
    // I blank final DEVONO essere inizializzati nel
    // costruttore:
    public BlankFinal() {
        j = 1; // Inizializza un blank final
        p = new Poppet(1); // Inizializza un riferimento blank
                           // final
    }
    public BlankFinal(int x) {
        j = x; // Inizializza un blank final
        p = new Poppet(x); // Inizializza un riferimento blank
                           // final
    }
    public static void main(String[] args) {
        new BlankFinal();
        new BlankFinal(47);
    }
} //:~
```

Java vi obbliga a eseguire le assegnazioni a **final**, tramite un'espressione nel punto di definizione del campo oppure in ogni costruttore, garantendo in tal modo che il campo **final** sia sempre inizializzato prima del suo utilizzo.

7 • Riutilizzo delle classi

Esercizio 19 (2) Create una classe con un riferimento blank **final** a un oggetto ed eseguite l'inizializzazione del blank **final** all'interno di tutti i costruttori. Dimostrate che Java garantisce l'inizializzazione del **final** prima dell'utilizzo e che dopo l'inizializzazione esso non può essere modificato.

Gli argomenti **final**

Java permette di rendere **final** gli argomenti dichiarandoli come tali nell'elenco degli argomenti. Questo significa che all'interno del metodo non potrete modificare ciò a cui punta il riferimento dell'argomento.

```
//: reusing/FinalArguments.java
// Utilizzo di "final" con gli argomenti dei metodi.

class Gizmo {
    public void spin() {}
}

public class FinalArguments {
    void with(final Gizmo g) {
        //! g = new Gizmo(); // Illegale -- g e' final
    }
    void without(Gizmo g) {
        g = new Gizmo(); // OK -- g non e' final
        g.spin();
    }
    // void f(final int i) { i++; } // Non puo' essere cambiato
    // Potete soltanto leggere da un primitivo final:
    int g(final int i) { return i + 1; }
    public static void main(String[] args) {
        FinalArguments bf = new FinalArguments();
        bf.without(null);
        bf.with(null);
    }
} //:~
```

I metodi **f()** e **g()** mostrano quanto accade quando gli argomenti primitivi sono **final**: potete leggere l'argomento ma non modificarlo. Questa caratteristica è utilizzata soprattutto per passare i dati a classi interne anonime, argomento che vedrete in dettaglio nel capitolo dedicato alle classi interne.

Metodi di tipo final

I motivi che giustificano l'esistenza di metodi **final** sono due. Il primo è applicare un "blocco" al metodo per evitare che qualunque classe derivata ne modifichi il significato: questa è una scelta progettuale, necessaria per assicurarsi che il comportamento di un metodo sia conservato dall'ereditarietà e non possa essere sovrascritto.

Il secondo motivo per cui in passato era raccomandato l'utilizzo dei metodi **final** è l'efficienza. Nelle precedenti implementazioni di Java, il fatto di rendere **final** un metodo permetteva al compilatore di trasformare qualsiasi chiamata a quel metodo in chiamata di tipo inline. Quando il compilatore trovava una chiamata a un metodo **final**, poteva decidere di ignorare l'approccio standard che prevedeva l'inclusione di codice adatto per eseguire il meccanismo di chiamata del metodo (inserimento degli argomenti in stack, salto al codice del metodo e sua esecuzione, salto al punto precedente e cleanup degli argomenti di stack, gestione del valore di ritorno), sostituendo invece la chiamata del metodo con una copia del codice effettivo presente nel corpo del metodo. L'utilizzo di **final** ha eliminato la necessità di chiamata. Naturalmente, con metodi di grandi dimensioni il vostro codice inizierebbe a "gonfiarsi" e probabilmente non otterreste alcun beneficio in termini di prestazioni dall'*Inlining*, poiché qualsiasi miglioramento sarebbe annullato dal tempo speso all'interno del metodo.

Nelle versioni più recenti di Java, in particolare grazie alle tecnologie hot-spot, la macchina virtuale può rilevare queste situazioni e ottimizzare l'indirezione supplementare (*indirection*); non è quindi più necessario ricorrere a **final** per cercare di "aiutare" l'ottimizzatore, anzi, è generalmente sconsigliato farlo. Con Java SE5/6 dovreste lasciare la gestione dei problemi di efficienza al compilatore e alla JVM e rendere un metodo **final** soltanto se volete esplicitamente evitare la sovrascrittura.¹

Java storia

1. Non lasciatevi prendere dall'urgenza di ottimizzare con troppo anticipo! Se la vostra applicazione è eccessivamente lenta, è improbabile che riusciate a risolvere il problema con la parola chiave **final**. All'indirizzo <http://mindview.net/Books/BetterJava> troverete informazioni sul profiling, una tecnica che può rivelarsi utile per velocizzare le applicazioni.

final e private

Ogni metodo **private** in una classe è implicitamente **final**: dal momento che non è permesso accedere a un metodo **private**, infatti, non si può neppure sovrascriverlo.

Potete aggiungere il modificatore **final** a un metodo **private**, ma questo non conferirebbe al metodo alcun significato aggiuntivo.

Questo argomento può provocare una certa confusione poiché, se provate a sovrascrivere un metodo **private** (che è implicitamente **final**), sembra funzionare e il compilatore non genererà alcun messaggio di errore.

```
//: reusing/FinalOverridingIllusion.java
// Sembra che sia possibile sovrascrivere
// un metodo private o un metodo private final.
import static net.mindview.util.Print.*;

class WithFinals {
    // Identico al solo "private":
    private final void f() { print("WithFinals.f()"); }
    // Anche automaticamente "final":
    private void g() { print("WithFinals.g()"); }
}

class OverridingPrivate extends WithFinals {
    private final void f() {
        print("OverridingPrivate.f()");
    }
    private void g() {
        print("OverridingPrivate.g()");
    }
}

class OverridingPrivate2 extends OverridingPrivate {
    public final void f() {
        print("OverridingPrivate2.f()");
    }
    public void g() {
```

```

        print("OverridingPrivate2.g()");
    }

}

public class FinalOverridingIllusion {
    public static void main(String[] args) {
        OverridingPrivate2 op2 = new OverridingPrivate2();
        op2.f();
        op2.g();
        // E' possibile eseguire l'upcast:
        OverridingPrivate op = op2;
        // Ma non e' possibile chiamare i metodi:
        //! op.f();
        //! op.g();
        // Anche qui:
        WithFinals wf = op2;
        //! wf.f();
        //! wf.g();
    }
} /* Output:
OverridingPrivate2.f()
OverridingPrivate2.g()
*///:~

```

La sovrascrittura (*overriding*) può verificarsi soltanto su un oggetto appartenente all’interfaccia della superclasse. In pratica deve essere possibile eseguire l’upcast sull’oggetto per convertirlo nel suo tipo di base e chiamare lo stesso metodo: il significato di questa condizione sarà chiarito nel prossimo capitolo. Se un metodo è **private** non fa parte dell’interfaccia della classe di base: si tratta di codice “nascosto” dentro la classe che si ritrova “per caso” ad avere quel nome particolare; quando create un metodo **public**, **protected** o con accesso di tipo package nella classe derivata, non realizzate alcun collegamento a un omonimo metodo nella classe di base. In questo caso non avrete sovrascritto il metodo, ma ne avete semplicemente creato uno nuovo. Poiché un metodo **private** è irraggiungibile ed è effettivamente invisibile, si esplicita soltanto nell’organizzazione del codice della classe per cui è stato definito, ossia può esistere soltanto in quel punto e non ha alcun collegamento con il metodo **private** della superclasse.

Esercizio 20 (1) Dimostrate che l’annotazione `@Override` risolve il problema illustrato in questo paragrafo.

Esercizio 21 (1) Create una classe con un metodo **final**, poi generate un’altra classe che eredita dalla prima e cercate di sovrascrivere quel metodo.

Classi di tipo final

Quando specificate **final** per un’intera classe, facendo precedere la sua definizione da questa parola chiave, indicate di non volere ereditare da questa classe né consentire che altri lo facciano: in altre parole, o la progettazione della vostra classe è tale che non avete mai bisogno di eseguire modifiche, oppure per motivi di sicurezza non volete che vengano create sottoclassi.

```

//: reusing/Jurassic.java
// Come rendere final un'intera classe.

class SmallBrain {}

final class Dinosaur {
    int i = 7;
    int j = 1;
    SmallBrain x = new SmallBrain();
    void f() {}
}

//! class Further extends Dinosaur {}
// errore: e' impossibile estendere la classe final 'Dinosaur'

public class Jurassic {
    public static void main(String[] args) {
        Dinosaur n = new Dinosaur();
        n.f();
        n.i = 40;
        n.j++;
    }
} ///:~

```

Notate che i campi di una classe **final** possono essere **final** oppure no, come preferite; le stesse regole si applicano ai campi **final**, a prescindere se la classe è stata definita come **final**. Tuttavia, poiché questo impedisce l'ereditarietà, tutti i metodi in una classe **final** sono anch'essi implicitamente **final**, dal momento che non esiste modo per sovrascriverli. Potete aggiungere il modificatore **final** a un metodo in una classe **final**, ma non aggiungerete alcun significato.

Esercizio 22 (1) Create una classe **final** e cercate di ereditare da essa.

Importanti considerazioni su final

In fase di progettazione di una classe potrebbe sembrarvi sensato rendere **final** un metodo, poiché potreste ritenere che nessuno avrà necessità di sovrascrivere i vostri metodi. Talvolta è così.

Prestate attenzione a formulare ipotesi affrettate, però: è difficile prevedere come una classe potrà essere riutilizzata, specialmente una di uso comune. Se definite un metodo come **final** potreste impedire il riutilizzo della vostra classe per ereditarietà nel progetto di altri programmatori, semplicemente perché non avete pensato che potesse essere utilizzata in quel modo.

La libreria standard Java è un eccellente esempio di questo concetto. In particolare, la classe Java 1.0/1.1 chiamata **Vector** è stata utilizzata diffusamente ma avrebbe potuto rivelarsi ancora più utile se, in nome di un'efficienza quasi certamente illusoria, tutti i metodi non fossero stati resi **final**. È concepibile che possiate voler ereditare e sovrascrivere una classe così fondamentale, tuttavia i progettisti hanno ritenuto che ciò non fosse appropriato. La situazione, in un certo senso, è ironica per due motivi. In primo luogo **Stack** deriva da **Vector**: ciò equivale ad affermare che uno **Stack** è un **Vector**, fatto non del tutto vero da un punto di vista logico; nondimeno, è un caso in cui i progettisti Java stessi hanno ereditato da **Vector**. Nel momento in cui hanno creato **Stack** in questo modo avrebbero dovuto rendersi conto che i metodi **final** erano troppo restrittivi.

In secondo luogo, molti tra i metodi principali di **Vector**, quali **addElement()** ed **elementAt()**, sono **synchronized**. Come vedrete nel capitolo dedicato alla concorrenza, questa condizione comporta oneri notevoli in termini di prestazioni, che probabilmente annullano qualsiasi beneficio fornito da **final**. Questo ha portato a formulare la teoria secondo la quale i programmatori sono puntualmente incapaci di “indovinare” dove dovrebbero avvenire le ottimizzazioni. È un peccato che una progettazione così grossolana abbia coinvolto la libreria standard, con cui ogni programmatore si ritrova a lavorare quotidianamente. Per fortuna la nuova libreria di contenitori Java sostituisce

Vector con **ArrayList**, che ha un comportamento decisamente migliore; tenete presente, però, che una parte considerevole del nuovo codice viene ancora scritto utilizzando la vecchia libreria.

È anche interessante osservare che **Hashtable**, un'altra classe importante della libreria standard 1.0/1.1, è priva di metodi **final**. Come già citato, appare evidente che alcune classi sono state progettate da persone diverse: noterete che i nomi dei metodi in **Hashtable** sono molto più brevi di quelli in **Vector**, un altro elemento a sostegno di questa tesi. Questo è precisamente il tipo di situazione che non dovrebbe risultare evidente agli utenti di una libreria di classi. La mancanza di coerenza si traduce in maggiore lavoro per i programmati: un'altra conferma del vantaggio di una progettazione e di un codice completi e trasparenti. Tenete presente che nella nuova libreria di contenitori Java **Hashtable** è stato sostituito da **HashMap**.

Inizializzazione e caricamento delle classi

Nei linguaggi più tradizionali i programmi vengono caricati in un unico momento, come parte della procedura di avvio; al caricamento segue l'inizializzazione, quindi il programma inizia. Il processo di inizializzazione in questi linguaggi deve essere studiato con attenzione, in modo che l'ordine in cui vengono inizializzati gli elementi **static** non generi malfunzionamenti. Il linguaggio C++, per esempio, dà problemi quando un oggetto **static** si aspetta che un altro **static** sia valido prima che il secondo oggetto sia stato inizializzato.

Java non presenta questo tipo di inconveniente poiché adotta un approccio diverso nella fase di caricamento: questa è una delle attività che vengono semplificate dal fatto che in Java tutto è un oggetto. Ricordate che il codice compilato per ogni classe si trova in un file separato, e che questo file non viene caricato fino a quando il codice non è necessario. In generale si può affermare che il codice della classe è caricato in occasione del suo primo utilizzo: questo corrisponde di solito al momento in cui viene costruito il primo oggetto di quella classe, ma il caricamento si verifica anche quando il programma accede a un campo o metodo di tipo **static**.²

Il momento o punto di primo utilizzo è anche quello in cui ha luogo l'inizializzazione statica. Al punto di caricamento tutti gli oggetti **static** e il blocco di codice **static** saranno inizializzati in “ordine testuale”, ovvero nell'ordine

2. Il costruttore è anch'esso un metodo **static**, benché tale parola chiave non sia indicata in modo esplicito. Più esattamente, il primo caricamento di una classe avviene quando il programma accede a uno qualsiasi dei suoi membri **static**.



in cui sono stati scritti nella definizione di classe; ovviamente, gli oggetti **static** vengono inizializzati una sola volta.

Inizializzazione con ereditarietà

Per una panoramica del processo di inizializzazione vi sarà utile analizzare l'intero processo, inclusa l'ereditarietà. Considerate l'esempio seguente:

```
//: reusing/Beetle.java
// Processo di inizializzazione completo.
import static net.mindview.util.Print.*;

class Insect {
    private int i = 9;
    protected int j;
    Insect() {
        print("i = " + i + ", j = " + j);
        j = 39;
    }
    private static int x1 =
        printInit("static Insect.x1 initialized");
    static int printInit(String s) {
        print(s);
        return 47;
    }
}

public class Beetle extends Insect {
    private int k = printInit("Beetle.k initialized");
    public Beetle() {
        print("k = " + k);
        print("j = " + j);
    }
    private static int x2 =
        printInit("static Beetle.x2 initialized");
    public static void main(String[] args) {
        print("Beetle constructor");
    }
}
```

```
Beetle b = new Beetle();
}
} /* Output:
static Insect.x1 initialized
static Beetle.x2 initialized
Beetle constructor
i = 9, j = 0
Beetle.k initialized
k = 47
j = 39
*///:~
```

La prima cosa che avviene eseguendo Java su **Beetle** è il tentativo di accedere a **Beetle.main()**, un metodo **static**, per far sì che il programma di caricamento recuperi il codice compilato per la classe **Beetle** (nel file chiamato **Beetle.class**). Durante il suo caricamento, il programma *loader* rileva che **Beetle** è una classe di base, indicata dalla parola chiave **extends**, e la carica come tale. Questo avverrà a prescindere se create o meno un oggetto da quella classe di base: verificatelo, commentando la creazione dell'oggetto.

Se la classe di base avesse a sua volta una propria classe di base, questa seconda superclasse sarebbe caricata anch'essa e così via. In seguito è eseguita l'inizializzazione statica nella classe di base radice (in questo caso, **Insect**), poi l'inizializzazione della successiva classe derivata e via di questo passo. Questa sequenza è importante, poiché l'inizializzazione **static** della classe derivata potrebbe dipendere dalla corretta inizializzazione del membro della classe di base.

A questo punto le classi necessarie sono state tutte caricate e l'oggetto può quindi essere creato: innanzitutto, a tutti i primitivi di questo oggetto sono assegnati i relativi valori predefiniti e i riferimenti agli oggetti vengono impostati a **null**; ciò avviene in un'unica operazione, ponendo a zero binario la memoria dell'oggetto. Poi viene chiamato il costruttore della classe di base: in questo caso specifico la chiamata è automatica, ma potete anche specificare la chiamata al costruttore della classe di base (come prima operazione nel costruttore di **Beetle()**) mediante la parola chiave **super**. Il costruttore della superclasse è sottoposto allo stesso processo del costruttore della classe derivata, nello stesso ordine; terminata questa operazione vengono inizializzate le variabili di istanza, in ordine testuale. Infine, viene eseguito il rimanente codice nel corpo del costruttore.

Esercizio 23 (2) Dimostrate che il caricamento di una classe ha luogo soltanto una volta e che può essere provocato sia dalla creazione della prima istanza di quella classe sia dall'accesso di un membro **static**.

Esercizio 24 (2) In **Beetle.java** ereditate un tipo specifico di oggetto-beetle dalla classe **Beetle**, adottando lo stesso schema delle classi esistenti. Analizzatene il funzionamento e commentate l'output.

Riepilogo

Ereditarietà e composizione sono due tecniche che permettono di creare nuovi tipi da quelli esistenti: la composizione riutilizza i tipi esistenti come parte dell'implementazione del nuovo tipo, mentre l'ereditarietà riutilizza l'interfaccia.

Nell'ereditarietà, la classe derivata adotta la stessa interfaccia della classe di base, in modo da poter essere sottoposta a upcast verso quest'ultima: un'operazione critica in sede di polimorfismo, come vedrete nel prossimo capitolo.

Malgrado la forte enfasi che viene data all'ereditarietà nella programmazione a oggetti, in sede di progettazione dovreste scegliere di preferenza la tecnica di composizione (o se possibile la delega) e servirvi dell'ereditarietà solo quando sia evidentemente necessario: tenete presente che la composizione è spesso la tecnica più flessibile. Tramite il meccanismo dell'ereditarietà con il vostro tipo di membro potete modificare in fase di esecuzione il tipo esatto, e quindi il comportamento, degli oggetti membro inseriti mediante composizione: avete quindi modo di intervenire sul comportamento dell'oggetto composto in fase di esecuzione.

Quando progettate un'applicazione il vostro obiettivo è trovare o creare un insieme di classi in cui ogni classe abbia un uso specifico, e non sia né troppo grande (con un numero così elevato di funzionalità da risultare ingombrante da riutilizzare) né eccessivamente piccola (da non essere utilizzabile così com'è o con l'integrazione di altre funzionalità). Se il progetto è troppo complesso, un metodo per renderlo più chiaro e gestibile consiste nel suddividere funzionalità complesse in altre più semplici: in tal caso l'errore del progettista è avere inserito troppe funzionalità in un numero di oggetti insufficiente.

Quando si progetta un'applicazione è essenziale rendersi conto che lo sviluppo di un programma, esattamente come la cultura umana, è un processo incrementale basato sulla sperimentazione: potete prostrarre l'analisi per quanto tempo volete, ma all'inizio del progetto ancora non avrete tutte le risposte che occorrono. Avrete più successo e un riscontro più immediato iniziando a

"coltivare" il progetto come se si trattasse di una creatura organica ed evolutiva, anziché costruirlo in un'unica soluzione come si fa con i grattacieli. L'ereditarietà e la composizione sono due strumenti fondamentali nella programmazione a oggetti che permettono di eseguire tale sperimentazione.

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



Capitolo 8

Polimorfismo

"Mi è stato chiesto: 'Mi scusi Signor Babbage, ma se si introducono valori sbagliati nella macchina, le risposte ottenute saranno giuste?'. Non sono neppure in grado di immaginare quale confusione di idee possa essere all'origine di una simile domanda."

Charles Babbage (1791-1871)



Dopo l'astrazione e l'ereditarietà, il polimorfismo è la terza caratteristica essenziale di ogni linguaggio di programmazione a oggetti e fornisce un'altra dimensione di separazione tra interfaccia e implementazione, permettendo di distinguere il *cosa* dal *come*.

Il polimorfismo consente una migliore organizzazione e leggibilità del codice, nonché la creazione di programmi estendibili che possono “crescere” non soltanto durante la creazione del progetto originale, ma anche quando sono richieste nuove funzionalità.

L'incapsulamento crea nuovi tipi di dato combinando caratteristiche e comportamenti; l'occultamento dell'implementazione separa da essa l'interfaccia, rendendo i suoi dettagli **private**. Questo tipo di organizzazione meccanica ha immediatamente senso per chi possiede un'esperienza di programmazione procedurale, ma il polimorfismo ha a che fare con la “dissociazione” (*decoupling*) in termini di tipi. Nel capitolo precedente avete visto che l'ereditarietà permette di considerare un oggetto come il proprio tipo o il suo tipo di base, una capacità determinante poiché consente di gestire molti tipi, derivati dallo stesso tipo di base, come se si trattasse di uno solo, e fa sì che lo stesso codice funzioni in modo coerente su tutti i tipi derivati. La chiamata di metodo polimorfa consente a un determinato tipo di esprimere la sua distinzione da un altro simile, purché entrambi siano derivati

dallo stesso tipo di base: questa distinzione si esprime mediante differenze nel comportamento dei metodi richiamabili attraverso la classe di base.

In questo capitolo imparerete a conoscere il polimorfismo (chiamato anche *binding dinamico*, *late binding* o *runtime binding*) partendo dalle nozioni di base, grazie a semplici esempi essenziali focalizzati unicamente sul comportamento polimorfo del programma.

Revisione dell'upcasting

Nel capitolo precedente avete visto che un oggetto può essere utilizzato come tipo a sé o come oggetto del tipo di base. Come sapete, l'operazione che consiste nel trattare un riferimento a un oggetto come riferimento al suo tipo di base è chiamata *upcasting*, per il modo in cui sono rappresentate le alberature di ereditarietà, con la classe di base in alto.

Avete anche visto che si verifica un problema, illustrato nel seguente esempio sugli strumenti musicali.

Per prima cosa, tenuto conto che diversi esempi ne fanno uso, è utile creare un'enumerazione **Note** separata, in un pacchetto.

```
//: polymorphism/music/Note.java
// Note eseguibili sugli strumenti musicali.
package polymorphism.music;

public enum Note {
    MIDDLE_C, C_SHARP, B_FLAT; // Ecc.
} ///:~
```

La parola chiave **enum** è stata analizzata nel Capitolo 5.

In questo caso **Wind** è un tipo di **Instrument**, pertanto **Instrument** viene ereditato da **Wind**.

```
//: polymorphism/music/Instrument.java
package polymorphism.music;
import static net.mindview.util.Print.*;

class Instrument {
    public void play(Note n) {
```

```
    print("Instrument.play()");
}
}

///:~

//: polymorphism/music/Wind.java
package polymorphism.music;

// Gli oggetti Wind sono Instrument
// poiché hanno la loro stessa interfaccia:
public class Wind extends Instrument {
    // Ridefinisce il metodo dell'interfaccia:
    public void play(Note n) {
        System.out.println("Wind.play() " + n);
    }
} ///:~

//: polymorphism/music/Music.java
// Ereditarietà e upcasting.
package polymorphism.music;

public class Music {
    public static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        tune(flute); // Upcasting
    }
} /* Output:
Wind.play() MIDDLE_C
*///:~
```

Il metodo **Music.tune()** accetta un riferimento **Instrument** ma anche qualsiasi oggetto deriva da **Instrument**: potete verificare questo comportamento in **main()**, quando un riferimento **Wind** è passato a **tune()** senza richiedere il

casting. Ciò è accettabile: l'interfaccia in **Instrument** deve esistere in **Wind**, poiché **Wind** è ereditato da **Instrument**. L'upcasting da **Wind** a **Instrument** può “limitare” questa interfaccia, tuttavia non può ridurla a niente di meno dell'interfaccia completa verso **Instrument**.

Dimenticate il tipo di oggetto

Music.java potrebbe apparirvi un caso singolare: perché mai qualcuno dovrebbe dimenticare volutamente il tipo di un oggetto? Invece questo è quanto accade in seguito all'upcasting, un risultato decisamente più immediato rispetto a quello che si otterrebbe se **tune()** si limitasse a considerare un riferimento **Wind** come proprio argomento. Questo evidenzia un punto essenziale: in quest'ultima ipotesi avreste bisogno di scrivere un nuovo **tune()** per ogni tipo di **Instrument** nell'applicazione. Supponete di adottare questa seconda soluzione e aggiungete gli strumenti **Stringed** e **Brass**.

```
//: polymorphism/music/Music2.java
// Sovraccarico invece di upcasting.
package polymorphism.music;
import static net.mindview.util.Print.*;

class Stringed extends Instrument {
    public void play(Note n) {
        print("Stringed.play() " + n);
    }
}

class Brass extends Instrument {
    public void play(Note n) {
        print("Brass.play() " + n);
    }
}

public class Music2 {
    public static void tune(Wind i) {
        i.play(Note.MIDDLE_C);
    }

    public static void tune(Stringed i) {
```

```
i.play(Note.MIDDLE_C);
}

public static void tune(Brass i) {
    i.play(Note.MIDDLE_C);
}

public static void main(String[] args) {
    Wind flute = new Wind();
    Stringed violin = new Stringed();
    Brass frenchHorn = new Brass();
    tune(flute); // Nessun upcasting
    tune(violin);
    tune(frenchHorn);
}
} /* Output:
Wind.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
*///:~
```

Benché funzioni, la soluzione presenta un notevole inconveniente poiché costringe a scrivere metodi (specifici per il tipo) per ogni nuova classe **Instrument** aggiunta.

Questo non significa soltanto un maggiore lavoro di programmazione, ma anche che l'eventuale aggiunta di un nuovo metodo come **tune()** o di un nuovo tipo di **Instrument** risulterà più complessa; se a questo aggiungete il fatto che il compilatore non produrrà alcun messaggio di errore qualora dimenticiate di sovraccaricare uno dei vostri metodi, vi renderete conto che l'intero processo con i tipi diventa improponibile.

Sarebbe certamente più semplice poter realizzare un solo metodo che come argomento accetta la classe di base, anziché ognuna delle specifiche classi derivate: in altre parole, sarebbe pratico poter dimenticare l'esistenza di classi derivate e scrivere il codice come se fosse rivolto unicamente alla classe di base.

Questo è esattamente il risultato che il polimorfismo consente di ottenere. Tuttavia, la maggior parte dei programmati con esperienza di programmazione procedurale ha qualche difficoltà ad afferrare il funzionamento del polimorfismo.

Esercizio 1 (2) Create una classe **Cycle**, con le sottoclassi **Unicycle**, **Bicycle** e **Tricycle**, e dimostrate che con un'istanza di ogni tipo è possibile eseguire l'upcasting a **Cycle** mediante un metodo **ride()**.

La svolta

La difficoltà con **Music.java** si evidenzia eseguendo il programma. L'output è **Wind.play()**: è chiaramente l'output richiesto, tuttavia questo tipo di funzionamento non sembra sensato. Osservate il metodo **tune()**:

```
public static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
}
```

Esso riceve un riferimento a **Instrument**; quindi, come può il compilatore sapere che il riferimento **Instrument** in questo caso specifico punta a un **Wind** e non a un **Brass** o a uno **Stringed**? Il compilatore non può saperlo. Per comprendere a fondo l'argomento sarà utile esaminare l'argomento del *binding*.

Binding delle chiamate a metodi

L'azione di collegare una chiamata di metodo al corpo del metodo è detta *binding*; quando tale collegamento è eseguito prima dell'esecuzione del programma da parte del compilatore o dell'eventuale linker è chiamato *early binding* o *binding anticipato* (noto anche come *binding statico* o *a tempo di compilazione*). Potreste non avere mai sentito prima questo termine poiché i linguaggi procedurali non offrono opzioni di sorta: per esempio C ha un solo genere di chiamata di metodo, quella di tipo iniziale.

La parte che nel programma precedente rischia di confondere si riferisce appunto al *binding anticipato*, poiché il compilatore non può sapere quale metodo chiamare disponendo unicamente del riferimento **Instrument**.

La soluzione è il cosiddetto *late binding* o *binding ritardato* (noto anche come *binding dinamico* o *a tempo di esecuzione*), mediante il quale il collegamento avviene al momento dell'esecuzione in funzione del tipo di oggetto. Quando un linguaggio adotta il *late binding* deve poter usufruire di un meccanismo per determinare il tipo di oggetto in fase di esecuzione, e chiamare così il metodo appropriato: in pratica il compilatore continua a ignorare di quale tipo

di oggetto si tratti, tuttavia il meccanismo di chiamata individua e chiama il corpo di metodo corretto. Benché l'implementazione del binding ritardato sia variabile in base al linguaggio, è evidente che richiede sempre la presenza di determinate informazioni negli oggetti.

In Java tutti i metodi utilizzano il *late binding*, a meno che non siano di tipo **static** o **final**; tenete presente che i metodi **private** sono implicitamente **final**: questo significa che di solito non dovete prendere alcuna decisione sul tipo di collegamento adottato, poiché in automatico Java ricorre al binding ritardato.

Perché mai dovreste dichiarare un metodo **final**? Come avete visto nel capitolo precedente, questo accorgimento innanzitutto impedisce a chiunque di sovrascrivere quel metodo, inoltre, condizione forse più importante, “dissattiva” efficacemente il binding dinamico, o meglio indica al compilatore che il *late binding* non è necessario: questo permette al compilatore di generare codice leggermente più efficiente per le chiamate di metodo **final**. Tuttavia, nella maggior parte dei casi, **final** non avrà effetto sulle prestazioni del vostro programma, pertanto è opportuno che utiliziate questa parola chiave solo come scelta progettuale e non come tentativo per migliorare le prestazioni.

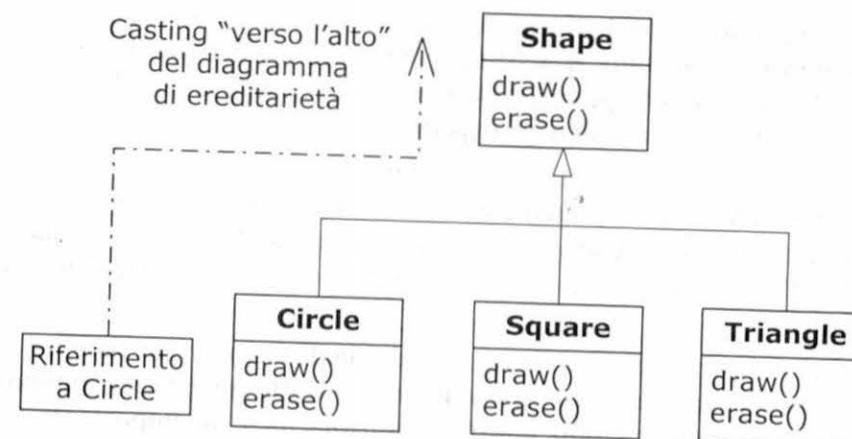
Come ottenere il comportamento corretto

Ora che sapete che tutti i collegamenti di metodo in Java avvengono in modo polimorfo per mezzo del *late binding* potete scrivere il codice che interagisce con la classe di base, con la certezza che tutte le possibili classi derivate funzioneranno correttamente utilizzando lo stesso codice: in altri termini, potrete inviare un messaggio a un oggetto e lasciare che questo scelga l'appoggio funzionale più adatto.

Un esempio classico nella programmazione a oggetti è quello delle forme, richiamato con frequenza poiché facile da visualizzare: potrebbe tuttavia confondere i programmatore principianti ritenere che la programmazione OOP si occupi soltanto di grafica, cosa evidentemente non vera.

L'esempio delle forme consiste di una classe di base chiamata **Shape** e diversi tipi derivati: cerchio (**Circle**), quadrato (**Square**), triangolo (**Triangle**) ecc.

La perfetta adeguatezza di questo esempio è motivata dalla facilità con cui è possibile affermare e comprendere, per esempio, che “un cerchio è un tipo di forma”. Il diagramma di ereditarietà a pagina seguente mostra le relazioni.



L'upcasting potrebbe verificarsi in un'istruzione semplice come

```
Shape s = new Circle();
```

che crea un oggetto **Circle** e assegna immediatamente il riferimento risultante a una forma **Shape**: all'apparenza sembrerebbe trattarsi di un errore (assegnazione di un tipo a un altro), tuttavia non lo è in quanto un **Circle** è una **Shape** per ereditarietà. Per questo motivo il compilatore accetta l'istruzione e non visualizza messaggi di errore.

Supponete di chiamare uno dei metodi della classe di base, che nelle classi derivate sono stati sovrascritti:

```
s.draw();
```

In questo caso, potreste aspettarvi che sia chiamato il metodo **draw()** di **Shape**, poiché dopotutto questo è un riferimento di **Shape**: quale motivo avrebbe il compilatore per chiamare qualcos'altro? Anche in questo caso, invece, è chiamato il metodo **Circle.draw()** corretto, grazie al late binding (polimorfismo).

L'esempio seguente si riferisce a una situazione leggermente diversa. Per prima cosa create una libreria riutilizzabile di tipi **Shape**:

```
//: polymorphism/shape/Shape.java
package polymorphism.shape;
import polymorphism.shape.*;
```

```

public class Shape {
    public void draw() {}
    public void erase() {}
} ///:~

//: polymorphism/shape/Circle.java
package polymorphism.shape;
import static net.mindview.util.Print.*;

public class Circle extends Shape {
    public void draw() { print("Circle.draw()"); }
    public void erase() { print("Circle.erase()"); }
} ///:~

//: polymorphism/shape/Square.java
package polymorphism.shape;
import static net.mindview.util.Print.*;

public class Square extends Shape {
    public void draw() { print("Square.draw()"); }
    public void erase() { print("Square.erase()"); }
} ///:~

//: polymorphism/shape/Triangle.java
package polymorphism.shape;
import static net.mindview.util.Print.*;

public class Triangle extends Shape {
    public void draw() { print("Triangle.draw()"); }
    public void erase() { print("Triangle.erase()"); }
} ///:~

//: polymorphism/shape/RandomShapeGenerator.java
// Una "factory" che genera forme in modo casuale.
package polymorphism.shape;
import java.util.*;
```



```

public class RandomShapeGenerator {
    private Random rand = new Random(47);
    public Shape next() {
        switch(rand.nextInt(3)) {
            default:
            case 0: return new Circle();
            case 1: return new Square();
            case 2: return new Triangle();
        }
    }
} //:~


//: polymorphism/Shapes.java
// Polimorfismo in Java.
import polymorphism.shape.*;

public class Shapes {
    private static RandomShapeGenerator gen =
        new RandomShapeGenerator();
    public static void main(String[] args) {
        Shape[] s = new Shape[9];
        // Riempie l'array di forme:
        for(int i = 0; i < s.length; i++)
            s[i] = gen.next();
        // Esegue le chiamate di metodo polimorfo:
        for(Shape shp : s)
            shp.draw();
    }
} /* Output:
Triangle.draw()
Triangle.draw()
Square.draw()
Triangle.draw()
Square.draw()
Triangle.draw()
Square.draw()

```



```

Triangle.draw()
Circle.draw()
*///:~

```

La classe di base **Shape** imposta l'interfaccia comune verso qualsiasi oggetto venga ereditato da **Shape**, consentendo in pratica di disegnare e cancellare qualsiasi forma. Le classi derivate sovrascrivono queste definizioni per assegnare un comportamento univoco a ogni specifico tipo di forma.

RandomShapeGenerator è una sorta di “fabbrica” (*factory*) che, ogni volta che chiamate il suo metodo **next()**, produce un riferimento a un oggetto **Shape** selezionato casualmente. Tenete presente che l'*upcasting* avviene nelle istruzioni **return**, ognuna delle quali riceve un riferimento a un **Circle**, uno **Square** o un **Triangle** che viene poi restituito da **next()** come tipo di ritorno, **Shape**. Con questa tecnica, ogni volta che viene chiamato **next()** non si è mai in grado di capire di quale tipo specifico si tratta, poiché viene sempre recuperato un riferimento a **Shape**.

Il metodo **main()** contiene un array di riferimenti **Shape** compilato tramite alcune chiamate a **RandomShapeGenerator.next()**. A questo punto sapete di avere alcune forme **Shape**, ma nessun altro dettaglio (come neppure il compilatore, del resto). Tuttavia, iterando l'array e chiamando **draw()** per ognuno dei suoi elementi, otterrete quasi “per magia” il giusto comportamento specifico per il tipo, com'è evidente dall'output del programma.

La creazione di forme in modo casuale consente di comprendere che il compilatore non dispone di nessuna informazione speciale che gli permetta di eseguire le chiamate corrette al momento della compilazione: tutte le chiamate a **draw()** devono avvenire tramite binding dinamico.

Esercizio 2 (1) Aggiungete l'annotazione **@Override** all'esempio delle forme.

Esercizio 3 (1) Aggiungete alla classe di base **Shapes.java** un nuovo metodo che visualizza un messaggio, ma senza sovrascriverlo nelle classi derivate: illustrate ciò che accade. Poi sovrascrivetelo in una delle classi derivate ma non nelle altre, e descrivete quanto accade; infine, sovrascrivetelo in tutte le classi derivate.

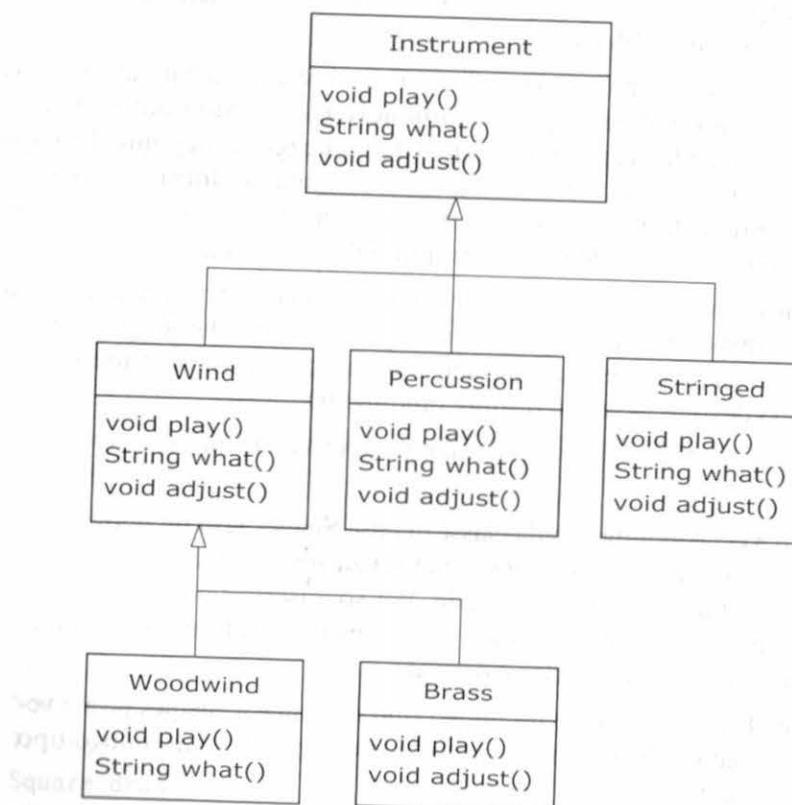
Esercizio 4 (2) Aggiungete un nuovo tipo di **Shape** a **Shapes.java** e verificate in **main()** che il polimorfismo funziona per il vostro nuovo tipo come nei vecchi.

Esercizio 5 (1) Partendo dall'Esercizio 1 aggiungete a **Cycle** un metodo **wheels()** che restituisce il numero di ruote. Modificate **ride()** in modo che chiama **wheels()** e verificate che il polimorfismo funziona.

Estensibilità

Ora ritornerete all'esempio degli strumenti musicali. Grazie al polimorfismo potete aggiungere all'applicazione quanti nuovi tipi desiderate, senza modificare il metodo `tune()`. In un programma a oggetti ben progettato, la maggior parte dei metodi seguirà il modello di `tune()` e comunicherà soltanto con l'interfaccia della classe di base. Un programma di questo tipo è estensibile, poiché consente di integrare nuove funzionalità ereditando nuovi tipi di dato dalla classe di base comune; inoltre, la gestione delle nuove classi non richiede modifiche ai metodi che manipolano l'interfaccia della classe di base.

Considerate quanto accade nell'esempio degli strumenti, aggiungendo altri metodi alla classe di base e un certo numero di nuove classi. Osservate il seguente diagramma.



Tutte queste nuove classi funzioneranno perfettamente con il vecchio metodo `tune()`, senza alcuna modifica. Nonostante `tune()` si trovi in un file separato e i nuovi metodi vengano aggiunti all'interfaccia di `Instrument`, il metodo `tune()` continuerà a funzionare in modo corretto senza richiedere una nuova compilazione. Quella che segue è l'implementazione del diagramma.

```

//: polymorphism/music3/Music3.java
// Un programma estensibile.
package polymorphism.music3;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

```

```

class Instrument {
    void play(Note n) { print("Instrument.play() " + n); }
    String what() { return "Instrument"; }
    void adjust() { print("Adjusting Instrument"); }
}

class Wind extends Instrument {
    void play(Note n) { print("Wind.play() " + n); }
    String what() { return "Wind"; }
    void adjust() { print("Adjusting Wind"); }
}

class Percussion extends Instrument {
    void play(Note n) { print("Percussion.play() " + n); }
    String what() { return "Percussion"; }
    void adjust() { print("Adjusting Percussion"); }
}

class Stringed extends Instrument {
    void play(Note n) { print("Stringed.play() " + n); }
    String what() { return "Stringed"; }
    void adjust() { print("Adjusting Stringed"); }
}
  
```



```

class Brass extends Wind {
    void play(Note n) { print("Brass.play() " + n); }
    void adjust() { print("Adjusting Brass"); }
}

class Woodwind extends Wind {
    void play(Note n) { print("Woodwind.play() " + n); }
    String what() { return "Woodwind"; }
}

public class Music3 {
    // Non tiene conto del tipo, quindi i nuovi tipi
    // aggiunti al sistema continuano a funzionare:
    public static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }

    public static void tuneAll(Instrument[] e) {
        for(Instrument i : e)
            tune(i);
    }

    public static void main(String[] args) {
        // Upcasting durante l'aggiunta all'array:
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
            new Brass(),
            new Woodwind()
        };
        tuneAll(orchestra);
    }
} /* Output:
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C

```

```

Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
*///:~

```

I nuovi metodi sono **what()**, che restituisce un riferimento **String** con una descrizione della classe, e **adjust()**, che accorda ogni strumento.

In **main()**, quando inserite uno strumento nell'array **orchestra** ne eseguite automaticamente l'upcast a **Instrument**.

Potete notare come il metodo **tune()** ignori le modifiche del codice e funzioni ancora regolarmente: questo è appunto il comportamento garantito dal polimorfismo.

Le modifiche al codice non danneggiano parti del programma che non dovrebbero essere interessate; in altre parole, il polimorfismo è una tecnica importante per il programmatore al fine di separare gli elementi che vengono alterati da quelli che rimangono invariati.

Esercizio 6 (1) Modificate **Music3.java** in modo che **what()** diventi il metodo **toString()** dell'oggetto radice **Object**. Provate a visualizzare gli oggetti **Instrument** utilizzando **System.out.println()** senza casting.

Esercizio 7 (2) Aggiungete un nuovo tipo di **Instrument** a **Music3.java** e verificate che il polimorfismo funzioni per il vostro nuovo tipo.

Esercizio 8 (2) Modificate **Music3.java** in modo che crei casualmente oggetti **Instrument** allo stesso modo di **Shapes.java**.

Esercizio 9 (3) Create una gerarchia ereditaria di roditori (**Rodent**): topo (**Mouse**), gerbillo (**Gerbil**), criceto (**Hamster**) ecc. Nella classe di base generate metodi comuni alla famiglia dei roditori, sovrascrivendo quelli che nelle classi derivate si riferiscono a comportamenti specifici delle singole specie. Create un array di roditori, completatelo con i diversi tipi di animali e chiamate i metodi della classe di base per verificare ciò che avviene.

Esercizio 10 (3) Create una classe di base con due metodi e nel primo chiamate il secondo; poi generate una classe ereditata da quella di base e sovrascrivete il secondo metodo, infine create un oggetto della classe derivata, eseguitene l'upcast al tipo di base e chiamate il primo metodo. Spiegate quello che accade.

Attenzione alla sovrascrittura dei metodi private!

Ecco qualcosa che potreste cercare di eseguire in buona fede.

```
//: polymorphism/PrivateOverride.java
// Tentativo di sovrascrittura di un metodo private.
package polymorphism;
import static net.mindview.util.Print.*;

public class PrivateOverride {
    private void f() { print("private f()"); }
    public static void main(String[] args) {
        PrivateOverride po = new Derived();
        po.f();
    }
}

class Derived extends PrivateOverride {
    public void f() { print("public f()"); }
} /* Output:
private f()
*///:~
```

Potreste ragionevolmente aspettarvi che l'output sia “**public f()**”: tuttavia un metodo **private** è automaticamente **final** e viene anche nascosto alla classe derivata. Pertanto il metodo **f()** di **Derived** in questo caso è nuovo di zecca; non è neppure sovraccarico, dal momento che la versione di **f()** della classe di base non è visibile a **Derived**.

Ne consegue che soltanto i metodi non **private** sono sovrascrivibili ed è quindi necessario prestare attenzione all'apparente sovrascrittura dei metodi **private**, per la quale il compilatore non genera alcun avvertimento, pur non assicurando il comportamento che vi aspettereste: in altri termini, nella classe derivata dovreste utilizzare un nome diverso da quello di un metodo **private** della classe di base.

Attenzione ai metodi e ai campi static!

Quando si viene a conoscenza dei meccanismi del polimorfismo si è portati a ritenere che esso coinvolga ogni aspetto della programmazione a oggetti,

tuttavia, soltanto le normali chiamate di metodo possono essere polimorfe. Per esempio, se accedete direttamente a un campo l'accesso sarà risolto in fase di compilazione, come dimostra il seguente esempio.¹

```
//: polymorphism/FieldAccess.java
/* L'accesso diretto ai campi viene determinato al momento
della compilazione.*/

class Super {
    public int field = 0;
    public int getField() { return field; }
}

class Sub extends Super {
    public int field = 1;
    public int getField() { return field; }
    public int getSuperField() { return super.field; }
}

public class FieldAccess {
    public static void main(String[] args) {
        Super sup = new Sub(); // Upcast
        System.out.println("sup.field = " + sup.field +
                           ", sup.getField() = " + sup.getField());
        Sub sub = new Sub();
        System.out.println("sub.field = " +
                           sub.field + ", sub.getField() = " +
                           sub.getField() +
                           ", sub.getSuperField() = " +
                           sub.getSuperField());
    }
} /* Output:
sup.field = 0, sup.getField() = 1
sub.field = 1, sub.getField() = 1, sub.getSuperField() = 0
*///:~
```

1. Si ringrazia Randy Nichols per aver proposto l'argomento.

Quando un oggetto **Sub** è sottoposto a upcast in un riferimento **Super**, qualsiasi accesso di campo viene risolto dal compilatore e non è quindi polimorfo. In questo esempio **Super.field** e **Sub.field** sono oggetto di due tecniche di allocazione diverse; di conseguenza **Sub** contiene in realtà due campi chiamati **field**: il proprio e quello che ottiene da **Super**. Tuttavia la versione **Super** non è quella predefinita che si produce quando vi riferite a **a field in Sub**; per ottenere il **field** di **Super** dovete indicare esplicitamente **super.field**.

Benché l'argomento possa apparire confuso, è virtualmente impossibile che questa situazione si verifichi nella pratica. Innanzitutto di norma renderete tutti i campi **private**, quindi inaccessibili direttamente ma soltanto come effetto collaterale delle chiamate ai metodi; inoltre, è ben difficile che assegniate lo stesso nome a un campo della classe di base e a un campo della classe derivata, poiché dareste luogo a un comportamento fuorviante.

Un metodo **static** non si comporta in modo polimorfo:

```
//: polymorphism/StaticPolimorphism.java
// I metodi static non sono polimorfi.

class StaticSuper {
    public static String staticGet() {
        return "Base staticGet()";
    }

    public String dynamicGet() {
        return "Base dynamicGet()";
    }
}

class StaticSub extends StaticSuper {
    public static String staticGet() {
        return "Derived staticGet()";
    }

    public String dynamicGet() {
        return "Derived dynamicGet()";
    }
}
```

Attenzione!

Ogni volta che crei un oggetto **Sub**, non avrai accesso al campo **staticGet()** della classe **StaticSub** perché sarà sovrascritto dal campo **staticGet()** della classe **StaticSuper**.

```
public class StaticPolimorphism {
    public static void main(String[] args) {
        StaticSuper sup = new StaticSub(); // Upcast
        System.out.println(sup.staticGet());
        System.out.println(sup.dynamicGet());
    }
} /* Output:
Base staticGet()
Derived dynamicGet()
*///:~
```

I metodi **static** sono infatti associati alla classe e non ai singoli oggetti.

Costruttori e polimorfismo

Come sempre, i costruttori sono diversi da altri tipi di metodo, e questo è vero anche nel caso del polimorfismo. Sebbene i costruttori non siano polimorfi (in effetti sono metodi statici, per i quali la dichiarazione **static** è implicita), è importante comprendere il loro funzionamento in gerarchie complesse e in abbinamento al polimorfismo: questa comprensione aiuterà a evitare situazioni spiacevoli.

Ordine delle chiamate al costruttore

L'ordine delle chiamate al costruttore è stato esaminato brevemente nel Capitolo 5 e di nuovo nel Capitolo 7, ma da punti di vista che ancora non tenevano conto del polimorfismo.

Un costruttore della classe di base viene sempre chiamato durante la costruzione di una classe derivata. Questa chiamata procede automaticamente verso l'alto nella gerarchia di ereditarietà, in modo che per ogni classe di base sia chiamato un costruttore: ciò ha senso poiché il costruttore ha il compito speciale di verificare che l'oggetto sia ben costruito. Poiché i campi sono generalmente di tipo **private**, si può dedurre che una classe derivata ha accesso soltanto ai propri membri, non a quelli della classe di base; il costruttore della classe di base è il solo a disporre delle informazioni e dei privilegi di accesso necessari per inizializzare i propri elementi. È quindi essenziale che tutti i costruttori siano chiamati, altrimenti l'intero oggetto non potrà essere costruito: è per questo motivo che il compilatore impone una chiamata di costruttore per ogni componente di una classe derivata. Nel caso non ven-

ga chiamato esplicitamente un costruttore della classe di base nel corpo del costruttore di quella derivata, il compilatore chiamerà in modo automatico il costruttore predefinito e, in assenza di quest'ultimo, segnalerà un errore. Tenete presente, infine, che qualora la classe non abbia alcun costruttore il compilatore ne creerà automaticamente uno predefinito.

Considerate l'esempio seguente, che mostra gli effetti di composizione, ereditarietà e polimorfismo sull'ordine di costruzione.

```
//: polymorphism/Sandwich.java
// Ordine delle chiamate al costruttore.
package polymorphism;
import static net.mindview.util.Print.*;

class Meal {
    Meal() { print("Meal()"); }
}

class Bread {
    Bread() { print("Bread()"); }
}

class Cheese {
    Cheese() { print("Cheese()"); }
}

class Lettuce {
    Lettuce() { print("Lettuce()"); }
}

class Lunch extends Meal {
    Lunch() { print("Lunch()"); }
}

class PortableLunch extends Lunch {
    PortableLunch() { print("PortableLunch()"); }
}
```

```
public class Sandwich extends PortableLunch {
    private Bread b = new Bread();
    private Cheese c = new Cheese();
    private Lettuce l = new Lettuce();
    public Sandwich() { print("Sandwich()"); }
    public static void main(String[] args) {
        new Sandwich();
    }
} /* Output:
Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwich()
*///:~
```

Questo codice crea una classe complessa composta da altre classi, ciascuna delle quali viene annunciata dal proprio costruttore. La classe importante è **Sandwich**, che riflette tre livelli di ereditarietà (quattro, considerando quella隐式的 da **Object**) e tre oggetti membro. Osservando l'output di un oggetto **Sandwich** creato in **main()** è possibile determinare che l'ordine delle chiamate ai costruttori per un oggetto complesso è quello indicato di seguito.

1. Chiamata al costruttore della classe di base. Questo passaggio si ripete ricorsivamente affinché la radice della gerarchia sia costruita per prima, seguita dalla successiva classe derivata e così via fino all'ultima classe derivata.
2. Chiamate agli inizializzatori di membro, nell'ordine in cui appaiono le relative dichiarazioni.
3. Chiamata al corpo del costruttore della classe derivata.

L'ordine delle chiamate ai costruttori è importante. Quando ereditate, sapete tutto sulla classe di base e potete accedere a qualsiasi metodo **public** e **protected** di questa classe: da questo è possibile dedurre che tutti i membri della classe di base sono validi anche nella classe derivata. In un metodo normale la costruzione ha già avuto luogo, pertanto i membri di tutti i componenti dell'oggetto sono stati costruiti; all'interno del costruttore, tuttavia, dovete avere la certezza che tutti i membri da utilizzare siano stati costruiti. L'unico

modo per garantire l'inizializzazione corretta è fare in modo che il costruttore della classe di base sia chiamato per primo. Quindi, quando vi trovate nel costruttore della classe derivata, tutti i membri ai quali potete accedere nella classe di base sono stati inizializzati. Sapere che tutti i membri all'interno del costruttore sono validi è anche lo scopo per cui, quando possibile, dovreste inizializzare tutti gli oggetti membro (ossia quelli inseriti nella classe mediante composizione) al loro punto di definizione nella classe, come **b**, **c** e **I** nell'esempio precedente: attenendovi a questa pratica contribuirete ad assicurare l'inizializzazione di tutti i membri della classe di base e degli oggetti membro dell'oggetto corrente. Sfortunatamente, però, questa tecnica non è adatta in ogni situazione, come vedrete nel prossimo paragrafo.

Esercizio 11(1) Aggiungete la classe **Pickle** (sottaceti) a **Sandwich.java**.

Ereditarietà e cleanup

Quando utilizzate la composizione e l'ereditarietà per creare una nuova classe non dovete quasi mai preoccuparvi del cleanup: di norma i sotto-oggetti possono essere lasciati al garbage collector.

Se avete specifiche esigenze di cleanup dovete creare un metodo **dispose()** per la vostra nuova classe: il termine *dispose* (eliminare) è stato scelto dall'autore, e certamente saprete scegliere un nome migliore. Con l'ereditarietà, inoltre, dovete sovraccaricare **dispose()** nella classe derivata, nel caso abbiate particolari esigenze di cleanup che devono essere eseguite durante la garbage collection. Quando sovraccaricate **dispose()** in una classe ereditata è importante che ricordiate di chiamare la versione **dispose()** della classe di base, altrimenti il cleanup di questa classe non avrà luogo. Questo è dimostrato nel seguente esempio.

```
//: polymorphism/Frog.java
// Cleanup ed ereditarietà.
package polymorphism;
import static net.mindview.util.Print.*;

class Characteristic {
    private String s;
    Characteristic(String s) {
        this.s = s;
        print("Creating Characteristic " + s);
    }
}
```

```
protected void dispose() {
    print("Disposing Characteristic " + s);
}
}

class Description {
    private String s;
    Description(String s) {
        this.s = s;
        print("Creating Description " + s);
    }
    protected void dispose() {
        print("Disposing Description " + s);
    }
}

class LivingCreature {
    private Characteristic p =
        new Characteristic("is alive");
    private Description t =
        new Description("Basic Living Creature");
    LivingCreature() {
        print("LivingCreature()");
    }
    protected void dispose() {
        print("LivingCreature dispose");
        t.dispose();
        p.dispose();
    }
}

class Animal extends LivingCreature {
    private Characteristic p =
        new Characteristic("has heart");
    private Description t =
        new Description("Animal not Vegetable");
}
```



```

Animal() { print("Animal()"); }
protected void dispose() {
    print("Animal dispose");
    t.dispose();
    p.dispose();
    super.dispose();
}
}

class Amphibian extends Animal {
    private Characteristic p =
        new Characteristic("can live in water");
    private Description t =
        new Description("Both water and land");
    Amphibian() {
        print("Amphibian()");
    }
    protected void dispose() {
        print("Amphibian dispose");
        t.dispose();
        p.dispose();
        super.dispose();
    }
}

public class Frog extends Amphibian {
    private Characteristic p = new Characteristic("Croaks");
    private Description t = new Description("Eats Bugs");
    public Frog() { print("Frog()"); }
    protected void dispose() {
        print("Frog dispose");
        t.dispose();
        p.dispose();
        super.dispose();
    }
}

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

```

```

Frog frog = new Frog();
print("Bye!");
frog.dispose();
}
} /* Output:
Creating Characteristic is alive
Creating Description Basic Living Creature
LivingCreature()
Creating Characteristic has heart
Creating Description Animal not Vegetable
Animal()
Creating Characteristic can live in water
Creating Description Both water and land
Amphibian()
Creating Characteristic Croaks
Creating Description Eats Bugs
Frog()
Bye!
Frog dispose
Disposing Description Eats Bugs
Disposing Characteristic Croaks
Amphibian dispose
Disposing Description Both water and land
Disposing Characteristic can live in water
Animal dispose
Disposing Description Animal not Vegetable
Disposing Characteristic has heart
Disposing LivingCreature
Disposing Description Basic Living Creature
Disposing Characteristic is alive
*///:~

```

Ogni classe nella gerarchia contiene anche oggetti membro dei tipi **Characteristic** e **Description**, anch'essi da eliminare. L'ordine di eliminazione dovrebbe essere contrario rispetto a quello di inizializzazione, nel caso un sotto-oggetto dipenda da un altro: per i campi equivale all'opposto dell'ordine di dichiarazione, dal momento che essi sono inizializzati in ordine di dichiarazione.

Secondo lo schema utilizzato in C++ per i distruttori, per le classi di base dovreste eseguire per primo il cleanup della classe derivata, poi quello della classe di base: questo per non rischiare di distruggere in anticipo metodi della classe di base (potenzialmente richiesti per il cleanup della classe derivata), il cui funzionamento potrebbe dipendere dalla disponibilità dei componenti della classe di base. L'output evidenzia che tutte le parti dell'oggetto **Frog** sono eliminate in ordine inverso di creazione.

Dall'esempio emerge chiaramente che, malgrado il cleanup non sia sempre necessario, quando lo eseguite dovete dedicargli tutta la vostra attenzione.

Esercizio 12 (3) Modificate l'Esercizio 9 in modo che dimostri l'ordine d'inizializzazione delle classi di base e delle classi derivate, poi aggiungete oggetti membro sia alla classe di base sia a quelle derivate e visualizzate il loro ordine d'inizializzazione durante la costruzione.

Noteate anche che, nell'esempio precedente, un oggetto **Frog** "possiede" i propri oggetti membro: li crea e ne stabilisce la durata (equivalente a quella di **Frog**), pertanto sa quando eliminare con **dispose()** questi oggetti. Tuttavia se uno di questi oggetti membro è condiviso con uno o più altri oggetti il problema diventerà più complesso e non potrete limitarvi a chiamare **dispose()**. In questi casi può essere necessario mantenere un conteggio dei riferimenti, allo scopo di tenere traccia del numero di oggetti che ancora accedono a un oggetto condiviso, come illustra il seguente esempio.

```
//: polymorphism/ReferenceCounting.java
// Cleanup di oggetti membro condivisi.
import static net.mindview.util.Print.*;

class Shared {
    private int refcount = 0;
    private static long counter = 0;
    private final long id = counter++;
    public Shared() {
        print("Creating " + this);
    }
    public void addRef() { refcount++; }
    protected void dispose() {
        if(--refcount == 0)
```

```
        print("Disposing " + this);
    }
    public String toString() { return "Shared " + id; }
}

class Composing {
    private Shared shared;
    private static long counter = 0;
    private final long id = counter++;
    public Composing(Shared shared) {
        print("Creating " + this);
        this.shared = shared;
        this.shared.addRef();
    }
    protected void dispose() {
        print("Disposing " + this);
        shared.dispose();
    }
    public String toString() { return "Composing " + id; }
}

public class ReferenceCounting {
    public static void main(String[] args) {
        Shared shared = new Shared();
        Composing[] composing = { new Composing(shared),
            new Composing(shared), new Composing(shared),
            new Composing(shared), new Composing(shared) };
        for(Composing c : composing)
            c.dispose();
    }
} /* Output:
Creating Shared 0
Creating Composing 0
Creating Composing 1
Creating Composing 2
Creating Composing 3
```

```

Creating Composing 4
Disposing Composing 0
Disposing Composing 1
Disposing Composing 2
Disposing Composing 3
Disposing Composing 4
Disposing Shared 0
*//:~

```

Il contatore **static long counter** tiene traccia del numero di istanze create per **Shared** e fornisce il valore di **id**. Il tipo di contatore è **long** invece di **int**, per evitare possibili *overflow*: negli esempi di questo volume il rischio è soltanto teorico, comunque sia è sempre opportuno adottare questa precauzione. L'**id** è **final** poiché il suo valore non deve cambiare per tutta la durata dell'oggetto.

Quando collegate un oggetto condiviso alla vostra classe dovete ricordare di chiamare **addRef()**, tuttavia il metodo **dispose()** terrà traccia del numero di riferimento e deciderà quando eseguire il cleanup. L'utilizzo di questa tecnica richiede un'attenzione supplementare, ma se condividerete oggetti che devono essere sottoposti a cleanup non avete alternativa.

Esercizio 13 (3) Aggiungete un metodo **finalize()** a **ReferenceCounting.java** per verificare la condizione di terminazione (Capitolo 7).

Esercizio 14 (4) Modificate l'Esercizio 12 in modo che uno degli oggetti membro sia condiviso e corredata di conteggio di riferimento e dimostratene il corretto funzionamento.

Comportamento dei metodi polimorfi all'interno dei costruttori

La gerarchia di chiamate ai costruttori propone un quesito interessante: che cosa accade se all'interno di un costruttore chiamate un metodo “dinamicamente collegato” dell'oggetto in costruzione, ossia risolto tramite binding dinamico?

In un metodo comune la chiamata dinamicamente collegata è risolta in fase di esecuzione, poiché l'oggetto non può sapere se appartiene alla classe in cui si trova il metodo oppure a qualche classe derivata da esso.

Se chiamate un metodo dinamicamente collegato all'interno di un costruttore, verrà anche utilizzata la definizione sovrascritta di quel metodo; tuttavia questo tipo di chiamata può produrre effetti inattesi poiché il metodo sovrascritto è chiamato prima del termine della costruzione dell'oggetto, una condizione che può generare bug di difficile individuazione.

Dal punto di vista concettuale il lavoro del costruttore è eseguire un insieme di operazioni che portano alla creazione dell'oggetto, un compito non sempre facile come sembra: all'interno di qualsiasi costruttore, infatti, l'oggetto potrebbe essere formato soltanto parzialmente, e Java permette di determinare soltanto che gli oggetti della classe di base sono stati inizializzati. Se il costruttore di una superclasse è una delle fasi della costruzione di un oggetto di una classe derivata, significa che i componenti derivati non sono ancora stati inizializzati nel momento in cui viene chiamato il costruttore della superclasse. Una chiamata di metodo dinamicamente collegata, tuttavia, “può estendersi” nella gerarchia di ereditarietà e chiamare un metodo in una classe derivata. Se adottate questo comportamento all'interno di un costruttore, potrete chiamare un metodo che manipola membri non ancora inizializzati: un trampolino diretto verso il disastro.

Il problema è descritto nell'esempio seguente.

```

//: polymorphism/PolyConstructors.java
// Costruttori e polimorfismo
// non producono cio' che vi aspettereste.
import static net.mindview.util.Print.*;

class Glyph {
    void draw() { print("Glyph.draw()"); }
    Glyph() {
        print("Glyph() before draw()");
        draw();
        print("Glyph() after draw()");
    }
}

class RoundGlyph extends Glyph {
    private int radius = 1;
    RoundGlyph(int r) {
        radius = r;
    }
}

```

```

    print("RoundGlyph.RoundGlyph(), radius = " + radius);
}

void draw() {
    print("RoundGlyph.draw(), radius = " + radius);
}

public class PolyConstructors {
    public static void main(String[] args) {
        new RoundGlyph(5);
    }

    /* Output:
    Glyph() before draw()
    RoundGlyph.draw(), radius = 0
    Glyph() after draw()
    RoundGlyph.RoundGlyph(), radius = 5
    */:~
}

```

Glyph.draw() è stato progettato per essere sovrascritto, operazione che avviene in **RoundGlyph**. Ma il costruttore di **Glyph** chiama questo metodo e la chiamata termina in **RoundGlyph.draw()**, il che sembra corrispondere allo scopo che si è prefisso il programmatore. Analizzando l'output, tuttavia, notate che quando il costruttore di **Glyph** chiama **draw()**, il valore del raggio non è neppure pari al valore predefinito iniziale (1): è 0. Questo probabilmente farà sì che sullo schermo venga disegnato un semplice punto o forse nulla, e voi rimarrete attoniti, cercando di immaginare perché il programma non funziona.

La chiave per risolvere il mistero è il fatto che l'ordine di inizializzazione descritto nel paragrafo precedente non è stato completato. L'effettivo processo di inizializzazione è il seguente.

1. Per prima cosa, la memoria allocata per l'oggetto viene inizializzata a zero binario.
2. I costruttori della classe di base vengono chiamati come descritto in precedenza. A questo punto è chiamato il metodo **draw()** sovrascritto (*prima* di chiamare il costruttore **RoundGlyph**), che rileva un valore di raggio pari a 0, a causa dell'inizializzazione avvenuta al Punto 1.
3. Gli inizializzatori dei membri vengono chiamati nell'ordine di dichiarazione.
4. Viene chiamato il corpo del costruttore della classe derivata.

Se non altro questa situazione ha un lato positivo, poiché tutto è stato inizializzato a 0 (qualsiasi cosa 0 rappresenti per quel tipo di dato specifico) e non semplicemente abbandonato al garbage collector: questo include riferimenti di oggetti che sono incorporati in una classe tramite la composizione, che assumono il valore **null**. Quindi, se dimenticate di inizializzare quel riferimento otterrete un'eccezione al momento dell'esecuzione, mentre tutto il resto risulterà a 0, un valore solitamente rivelatore in fase di controllo dell'output.

D'altra parte il risultato di questo programma dovrebbe avervi lasciati inorriditi, tenuto conto che avete eseguito un'operazione perfettamente logica che tuttavia ha prodotto un comportamento errato, senza alcun avvertimento da parte del compilatore: tenete presente che in questa situazione C++ rivela un comportamento assai più razionale. Errori di questo tipo potrebbero essere ben nascosti e richiedere molto tempo prima di essere scoperti.

Per queste ragioni, una raccomandazione di cui dovreste tenere conto per i costruttori è: "Fate il meno possibile per impostare l'oggetto in uno stato valido e se non potete farne a meno, evitate di chiamare altri metodi in quella classe". Gli unici metodi richiamabili senza problemi all'interno di un costruttore sono quelli definiti come **final** nella classe di base, inclusi i metodi **private**, che sono automaticamente **final**. Tali metodi non possono essere sovrascritti, quindi non offrono questo genere di sorprese. Non è detto che sia sempre possibile seguire questa direttiva, tuttavia cercate di farlo.

Esercizio 15 (2) Aggiungete una classe **RectangularGlyph** a **PolyConstructors.java** e dimostrate il problema descritto in questo paragrafo.

Tipi di ritorno covarianti

Java SE5 ha introdotto nuovi tipi di ritorno chiamati "covarianti" (*covariant return type*), grazie ai quali un metodo sovrascritto in una classe derivata può restituire un tipo derivato dal tipo restituito dal metodo della classe di base.

```
//: polymorphism/CovariantReturn.java
```

```

class Grain {
    public String toString() { return "Grain"; }
}

```



```

class Wheat extends Grain {
    public String toString() { return "Wheat"; }
}

class Mill {
    Grain process() { return new Grain(); }
}

class WheatMill extends Mill {
    Wheat process() { return new Wheat(); }
}

public class CovariantReturn {
    public static void main(String[] args) {
        Mill m = new Mill();
        Grain g = m.process();
        System.out.println(g);
        m = new WheatMill();
        g = m.process();
        System.out.println(g);
    }
} /* Output:
Grain
Wheat
*/

```

La differenza fondamentale tra Java SE5 e le versioni precedenti del linguaggio è che queste ultime forzerebbero la versione sovrascritta di **process()** in modo da restituire **Grain** e non **Wheat**, sebbene **Wheat** sia derivato da **Grain** e rappresenti quindi un tipo di ritorno del tutto legittimo. Grazie ai tipi di ritorno covarianti è invece possibile recuperare il tipo **Wheat**, più specifico.

Progettazione in funzione dell'ereditarietà

Quando venite a conoscenza del polimorfismo potreste ritenere che tutto debba essere ereditato, poiché il polimorfismo è un meccanismo decisamente funzionale. Questo orientamento potrebbe rendere i vostri progetti oltremo-

do pesanti: infatti, se nell'utilizzo di una classe esistente per costruirne una nuova scegliete come prima soluzione l'ereditarietà, il codice potrebbe diventare inutilmente complesso.

Un approccio migliore è scegliere in prima istanza la composizione, specialmente quando non è ancora evidente quale delle due ipotesi sia preferibile. La composizione non vi obbliga a impostare la progettazione su una gerarchia ereditaria ed è anche più flessibile nel permettervi di scegliere dinamicamente un tipo (e quindi un comportamento), mentre l'ereditarietà richiede di conoscere il tipo esatto al momento della compilazione. Questo concetto è illustrato nell'esempio seguente.

```

//: polymorphism/Transmogrify.java
// Modifica dinamica del comportamento di un oggetto
// tramite composizione (il design pattern "State").
import static net.mindview.util.Print.*;

class Actor {
    public void act() {}
}

class HappyActor extends Actor {
    public void act() { print("HappyActor"); }
}

class SadActor extends Actor {
    public void act() { print("SadActor"); }
}

class Stage {
    private Actor actor = new HappyActor();
    public void change() { actor = new SadActor(); }
    public void performPlay() { actor.act(); }
}

public class Transmogrify {
    public static void main(String[] args) {
        Stage stage = new Stage();

```

```

stage.performPlay();
stage.change();
stage.performPlay();
}
} /* Output:
HappyActor
SadActor
*///:~

```

Un oggetto **Stage** contiene un riferimento a un **Actor**, inizializzato a un oggetto **HappyActor**: questo vuole dire che **performPlay()** produce un determinato comportamento.

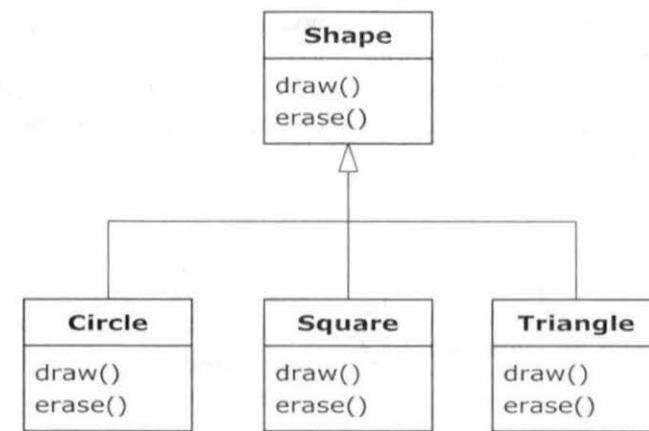
Tuttavia, poiché in fase di esecuzione un riferimento può essere collegato nuovamente a un altro oggetto, un riferimento di un oggetto **SadActor** può essere sostituito in **actor**, alterando il comportamento prodotto da **performPlay()**: in definitiva, questo si traduce in una maggiore flessibilità dinamica in fase di esecuzione. L'approccio descritto è noto come *State pattern*, sul quale troverete maggiori informazioni in *Thinking in Patterns (with Java)*, disponibile all'indirizzo www.mindview.net. Di contro non potete decidere di ereditare in modo diverso al momento dell'esecuzione: le condizioni di ereditarietà devono essere decise in fase di compilazione.

Una raccomandazione generica prevede che l'ereditarietà debba essere utilizzata per esprimere le differenze nel comportamento, e i campi per rappresentare variazioni di stato. L'esempio precedente adotta entrambe le tecniche: nel metodo **act()** vengono ereditate due classi con comportamenti diversi, mentre **Stage** utilizza la composizione per consentirvi di cambiare il suo stato. In questo caso, tale variazione di stato produce una modifica nel comportamento.

Esercizio 16 (3) Seguendo l'esempio in **Transmogrify.java** create una classe **Starship** contenente un riferimento **AlertStatus** che può indicare tre stati diversi. Includete metodi per modificare tali stati.

Confronto tra sostituzione ed estensione

Sembrerebbe che il modo più lineare per creare una gerarchia di ereditarietà consista nell'adottare l'approccio “puro”. In pratica, esso prevede che soltanto i metodi che sono stati impostati nella classe di base siano sovrascritti nella classe derivata, come mostra il diagramma seguente.



Questo comportamento può essere definito come relazione pura di tipo “è-un”, poiché è l'interfaccia di una classe che stabilisce ciò che essa rappresenta. L'ereditarietà garantisce che qualsiasi classe derivata non abbia niente di meno dell'interfaccia della classe di base; seguendo questo diagramma, però, noterete anche che le classi derivate non hanno niente di più dell'interfaccia della classe di base.

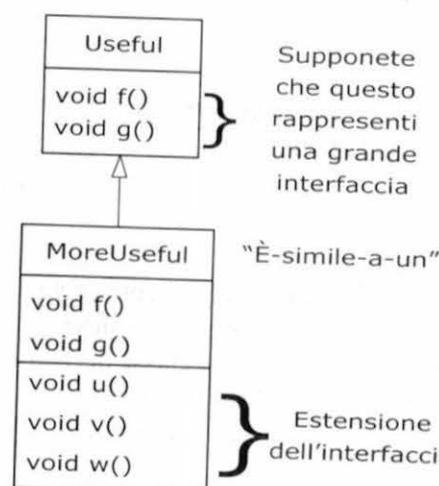
Questa può essere descritta come sostituzione pura, poiché gli oggetti delle classi derivate sono perfettamente sostituibili con la classe di base al momento del loro utilizzo, senza necessità di altre informazioni sulle sottoclassi.



In altri termini, la classe di base è in grado di ricevere qualsiasi messaggio che sia possibile inviare alla classe derivata, poiché le due classi hanno esattamente la stessa interfaccia. Tutto ciò che dovete fare è eseguire l'upcast dalla classe derivata, senza mai “voltarvi” per vedere l'esatto tipo di oggetto che state gestendo in quel momento: tutto è a carico del polimorfismo.

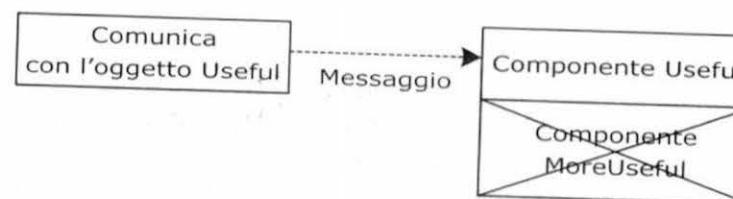
Se considerate le cose in questo modo vi sembrerà che una relazione pura di tipo “è-un” sia l'unico modo sensato di procedere e che qualsiasi altro approccio progettuale sia sinonimo di confusione e, per definizione, errato. Ma anche questa è una trappola. Non appena cominciate a ragionare in questi termini, scoprirete che l'ampliamento dell'interfaccia (che, sfortunatamente,

la parola chiave **extends** sembra incoraggiare) è la soluzione perfetta a un particolare problema. Tale relazione può essere definita come *è-simile-a-un*, poiché la classe derivata ha la stessa fondamentale interfaccia della classe di base, tuttavia offre altre caratteristiche che richiedono l'implementazione di metodi aggiuntivi.



A seconda delle situazioni questo è anche un approccio utile e sensato, tuttavia presenta un inconveniente.

Il componente esteso dell'interfaccia nella classe derivata non è disponibile alla classe di base, pertanto, una volta eseguito l'upcasting, non vi sarà possibile chiamare i nuovi metodi.



Se in questo caso non ricorrete all'upcasting non avrete problemi, ma vi ri troverete spesso ad avere bisogno di identificare il tipo di oggetto per accedere ai metodi estesi di quel tipo. Nel prossimo paragrafo vedrete come scoprire queste informazioni.

Downcasting e tipi di informazioni a runtime

Salendo di un livello nella gerarchia ereditaria l'upcasting provoca la perdita di specifiche informazioni di tipo, ed è perciò ragionevole pensare che il recupero di tali informazioni, ossia la discesa di un livello nella gerarchia, possa avvenire tramite il downcasting. Sapete che l'upcasting è sempre sicuro, visto che la classe di base non può avere un'interfaccia più estesa della classe derivata: è quindi indubbio che ogni messaggio inviato attraverso l'interfaccia della classe di base sarà accettato. Nel caso del downcasting, tuttavia, non potete avere la certezza che una forma, per esempio, sia effettivamente un cerchio, poiché potrebbe anche essere un triangolo, un quadrato o qualche altro tipo di oggetto.

Per risolvere questo problema deve esistere un modo che garantisca la correttezza di un downcasting, impedendovi di eseguire inadvertitamente un cast nel tipo errato e di inviare un messaggio che l'oggetto è incapace di accettare: un'operazione che sarebbe decisamente poco sicura.

In alcuni linguaggi, come il C++, per ottenere un downcasting di tipo sicuro dovete eseguire un'operazione speciale, mentre in Java ogni cast è controllato. Così, malgrado sembri un normale cast in parentesi, questo cast viene verificato in fase di esecuzione per assicurarvi che il tipo sia in effetti quello che vi aspettate: se così non fosse otterreste un errore **ClassCastException**. Questa azione di controllo dei tipi in fase di esecuzione è chiamata *“informazione dei tipi a runtime”* (RTTI, *RunTime Type Information*), e il suo comportamento è illustrato nell'esempio seguente.

```

//: polymorphism/RTTI.java
// Downcasting & RunTime Type Information (RTTI).
// {ThrowsException}

class Useful {
    public void f() {}
    public void g() {}
}

class MoreUseful extends Useful {
    public void f() {}
    public void g() {}
    public void u() {}
    public void v() {}
}
  
```



```
public void wO {}  
}  
  
public class RTTI {  
    public static void main(String[] args) {  
        Useful[] x = {  
            new Useful(),  
            new MoreUseful()  
        };  
        x[0].f();  
        x[1].g();  
        // In fase di compilazione: metodo non trovato in Useful:  
        //! x[1].u();  
        ((MoreUseful)x[1]).u(); // Downcast/RTTI  
        ((MoreUseful)x[0]).u(); // Solleva una eccezione  
    }  
} //:~
```

Come nel diagramma precedente, **MoreUseful** amplia l'interfaccia di **Useful**, ma poiché è ereditato può anche essere sottoposto a upcast a **Useful**. Nel codice precedente, questo accade nell'inizializzazione dell'array **x** in **main()**: poiché entrambi gli oggetti nell'array sono della classe **Useful** potete inviare i metodi **f()** e **g()** a entrambe le classi; se provate a chiamare **u()**, che esiste solo in **MoreUseful**, il compilatore visualizzerà un messaggio di errore.

Per accedere all'interfaccia estesa di un oggetto **MoreUseful** potete provare a eseguire un downcasting: se il tipo è corretto l'operazione avrà successo, altrimenti otterrete un errore **ClassCastException**. Non occorre scrivere nessun codice speciale per gestire questa eccezione, poiché segnala un errore di programmazione che potrebbe verificarsi ovunque in un programma. Il tag di commento **{ThrowsException}** indica al software di costruzione di questo manuale (Ant) che il programma **RTTI** potrebbe sollevare un'eccezione in fase di esecuzione.

Tuttavia, la tecnica RTTI è ben più di un semplice cast: per esempio, consente di determinare il tipo che state gestendo, prima di provare a eseguirne il downcasting. Nel Volume 2, tutto il Capitolo 2 è dedicato all'analisi dei vari tipi di informazioni RTTI di Java.

Esercizio 17 (2) Utilizzando la gerarchia **Cycle** dell'Esercizio 1, aggiungete un metodo **balance()** a **Unicycle** e **Bicycle**, ma non a **Tricycle**.

Create istanze dei tre tipi ed eseguitene l'upcast a un array di **Cycle**. Provate a chiamare **balance()** su ogni elemento dell'array e prendete nota dei risultati. Eseguite il downcast, chiamate **balance()** e osservate ciò che accade.

Riepilogo

Polimorfismo significa “varietà di forme”. Nella programmazione orientata agli oggetti la stessa interfaccia della classe di base è utilizzata in diverse forme: le varie versioni dei metodi dinamicamente collegati, ossia risolti tramite binding dinamico.

Questo capitolo ha mostrato che è impossibile capire, e creare, un esempio di polimorfismo senza ricorrere all'astrazione dei dati e all'ereditarietà. Il polimorfismo è una funzionalità che non può essere osservata in condizioni di isolamento ma solo in presenza di relazioni ereditarie: a differenza di una dichiarazione **switch**, per esempio, il polimorfismo funziona solo “di concerto”, come parte della visione globale delle relazioni di classe.

Per utilizzare efficacemente il polimorfismo nei vostri programmi, e quindi le tecniche orientate agli oggetti, dovete ampliare la vostra visione di programmatore per includervi non soltanto i membri e i messaggi di una singola classe, ma anche la condivisione di caratteristiche tra le classi e le loro reciproche relazioni. Nonostante richieda un notevole impegno è uno sforzo meritevole di essere intrapreso, i cui benefici possono essere riassunti in un ciclo di sviluppo più rapido, una migliore organizzazione del codice, l'ottenimento di programmi estensibili e una manutenzione semplificata del codice.

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

Le interfacce



Le interfacce e le classi astratte forniscono meccanismi più strutturati mediante i quali separare l'interfaccia dall'implementazione.

Tali meccanismi sono poco comuni nei linguaggi di programmazione: in C++, per esempio, queste nozioni sono supportate soltanto in modo indiretto. Il fatto che in Java siano state previste parole chiave specifiche indica che questi concetti sono stati considerati abbastanza importanti da esigere un supporto diretto.

In questo capitolo saranno esaminate le *classi astratte*, che costituiscono una sorta di compromesso tra le normali classi e le interfacce. Sebbene il primo impulso suggerisca la creazione di un'interfaccia, la classe astratta è comunque uno strumento importante, necessario per la costruzione di classi nelle quali non tutti i metodi siano stati implementati: non sempre è possibile ricorrere a un'interfaccia pura.

Metodi e classi astratti

In tutti gli esempi musicali del capitolo precedente i metodi della classe di base **Instrument** erano sempre fintizi: se questi metodi venissero effettivamente chiamati, significherebbe che c'è stato un errore! Infatti lo scopo della classe **Instrument** è creare un'interfaccia comune per tutte le classi derivate.

Negli esempi precedenti, l'unico motivo per impostare questa interfaccia comune è consentire alla classe di esprimersi in modo diverso per ogni sottotipo: essa imposta un modello di base, che specifica ciò che le classi derivate devono avere in comune. Un altro modo per definire questo modello è assegnare a **Instrument** il termine di *classe di base astratta*, o più semplicemente *classe astratta*.

In una classe astratta come **Instrument** gli oggetti della classe specifica non hanno quasi mai significato. Una classe astratta viene creata quando si desidera manipolare un insieme di classi tramite la sua interfaccia comune. Pertanto **Instrument** definisce soltanto l'interfaccia, non una particolare implementazione: di conseguenza la creazione di un oggetto di tipo **Instrument** non ha senso, al punto che probabilmente vorrete impedire all'utente di eseguire tale operazione.

Uno dei modi per farlo è provocare errori a fronte di chiamate a qualsiasi metodo di **Instrument**, tuttavia questo ritarderebbe le informazioni fino a momento dell'esecuzione e richiederebbe verifiche esaustive da parte del programmatore client. Di solito è più opportuno affrontare i problemi in fase di compilazione.

Java fornisce lo strumento adatto: il *metodo astratto*.¹

Concettualmente si tratta di un metodo incompleto, che possiede soltanto una dichiarazione ed è privo di corpo del metodo. La sintassi per dichiarare un metodo astratto è la seguente:

```
Abstract void f();
```

Una classe che contiene metodi astratti è chiamata *classe astratta*. Quando una classe contiene uno o più metodi astratti deve essere qualificata come **abstract**: in caso contrario il compilatore visualizzerà un messaggio di errore.

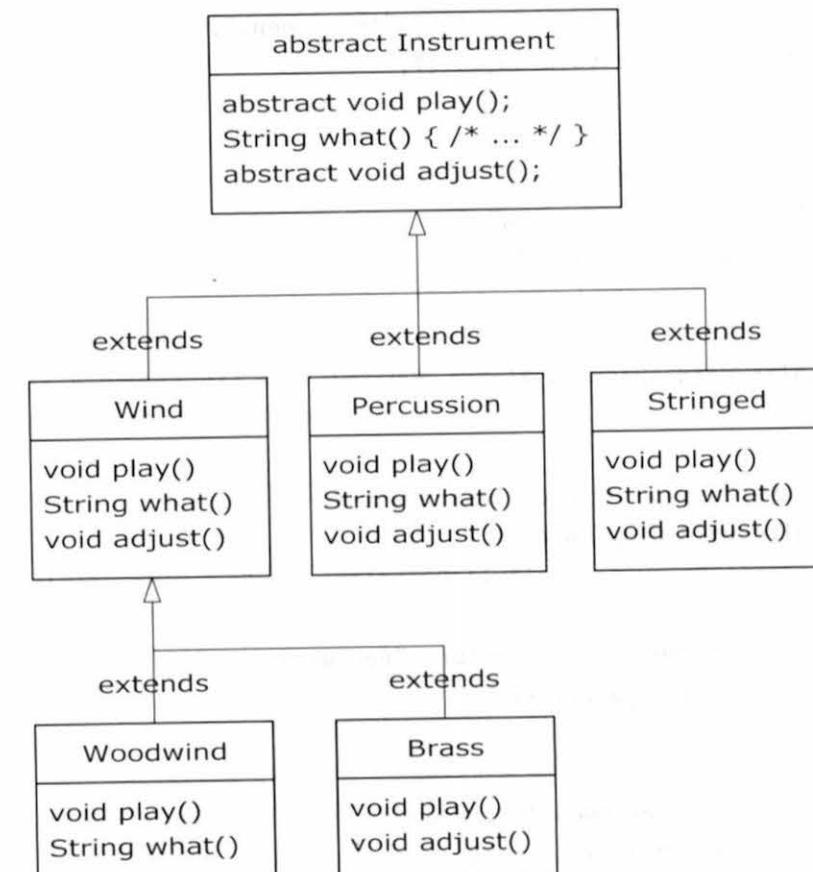
Se una classe astratta è incompleta, che cosa si suppone faccia il compilatore quando cercate di creare un oggetto di quella classe? Certamente non può creare un oggetto di una classe astratta, quindi produce un messaggio di errore. In questo modo il compilatore assicura la “purezza” della classe astratta e voi non dovete preoccuparvi di un suo eventuale uso improprio.

Se ereditate da una classe astratta e volete creare oggetti del nuovo tipo, dovrete fornire definizioni di metodo per tutti i metodi astratti presenti nella

classe di base. Qualora non eseguite questa operazione, che è opzionale, anche la classe derivata sarebbe astratta, e il compilatore vi costringerebbe a qualificare quella classe con la parola chiave **abstract**.

È possibile creare una classe **abstract** senza includervi alcun metodo astratto: una tecnica utile quando disponete di una classe in cui non abbia senso avere metodi **abstract**, e anche se volete evitare che ne vengano create istanze.

La classe **Instrument** del capitolo precedente può essere facilmente convertita in una classe astratta. Soltanto alcuni metodi saranno **abstract**, in quanto la creazione di una classe astratta non obbliga a rendere astratti tutti i metodi, e il risultato sarà il seguente.



1. Per i programmatori di C++, il metodo astratto corrisponde alla cosiddetta “funzione virtuale pura” (*pure virtual function*).



Ecco come appare l'esempio "musicale" modificato per tenere conto di classi e metodi **abstract**.

```
//: interfaces/music4/Music4.java
// Classi e metodi astratti.
package interfaces.music4;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

abstract class Instrument {
    private int i; // Allocazione di memoria
    public abstract void play(Note n);
    public String what() { return "Instrument"; }
    public abstract void adjust();
}

class Wind extends Instrument {
    public void play(Note n) {
        print("Wind.play() " + n);
    }
    public String what() { return "Wind"; }
    public void adjust() {}
}

class Percussion extends Instrument {
    public void play(Note n) {
        print("Percussion.play() " + n);
    }
    public String what() { return "Percussion"; }
    public void adjust() {}
}

class Stringed extends Instrument {
    public void play(Note n) {
        print("Stringed.play() " + n);
    }
}
```

```
public String what() { return "Stringed"; }
public void adjust() {}

class Brass extends Wind {
    public void play(Note n) {
        print("Brass.play() " + n);
    }
    public void adjust() { print("Brass.adjust()"); }
}

class Woodwind extends Wind {
    public void play(Note n) {
        print("Woodwind.play() " + n);
    }
    public String what() { return "Woodwind"; }
}

public class Music4 {
    // Non e' necessario preoccuparsi del tipo, quindi i nuovi
    // tipi
    // aggiunti all'applicazione funzionano senza problemi:
    static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }
    static void tuneAll(Instrument[] e) {
        for(Instrument i : e)
            tune(i);
    }
    public static void main(String[] args) {
        // Upcasting durante l'aggiunta all'array:
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
        }
```



```
    new Brass(),
    new Woodwind()
};

tuneAll(orchestra);

} /* Output:
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
*///:~
```

Potete constatare che le uniche modifiche interessano la classe di base.

Le classi e i metodi **abstract** sono utili poiché rendono esplicita l'astrattezza di una classe e in tal modo spiegano, sia all'utente sia al compilatore, come devono essere utilizzati.

Le classi astratte sono anche utili strumenti di *refactoring*, poiché permettono di spostare facilmente verso l'alto metodi comuni nella gerarchia di ereditarietà.

Esercizio 1 (1) Modificate l'Esercizio 9 del capitolo precedente in modo che **Rodent** sia una classe **abstract** e rendete astratti i metodi di questa classe ogni volta che sia possibile.

Esercizio 2 (1) Create una classe astratta senza includere metodi astratti, quindi verificate l'impossibilità di creare istanze di quella classe.

Esercizio 3 (2) Create una classe di base con un metodo **abstract print()** che viene sovrascritto in una classe derivata. La versione sovrascritta del metodo visualizza il valore di una variabile **int** definita nella classe derivata: quando definite la variabile assegnatele un valore diverso da zero. Nel costruttore della classe di base chiamate questo metodo; in **main()** create un oggetto del tipo derivato, poi chiamate il suo metodo **print()**. Commentate i risultati.

Esercizio 4 (3) Create una classe **abstract** priva di metodi, quindi una classe derivata e aggiungetevi un metodo. Generate un metodo **static** che accetta un riferimento alla classe di base, ne esegue il downcast alla classe derivata e chiama il metodo. In **main()**, dimostrate il funzionamento del progetto.

Ora inserite la dichiarazione **abstract** del metodo nella classe di base, eliminando così la necessità del downcast.

Interfacce

La parola chiave **interface** rappresenta un passo in avanti nel concetto di astrazione.

La parola chiave **abstract**, come sapete, permette di creare in una classe uno o più metodi indefiniti, allo scopo di mettere a disposizione una parte dell'interfaccia senza la corrispondente implementazione, che viene poi fornita dalle classi derivate.

La parola chiave **interface**, invece, crea una classe completamente astratta, vale a dire priva di implementazione, che consente al programmatore di stabilire esattamente i nomi dei metodi, gli elenchi di argomenti e i tipi di ritorno, ma senza corpi di metodo: un'interfaccia fornisce il solo schema, non l'implementazione.

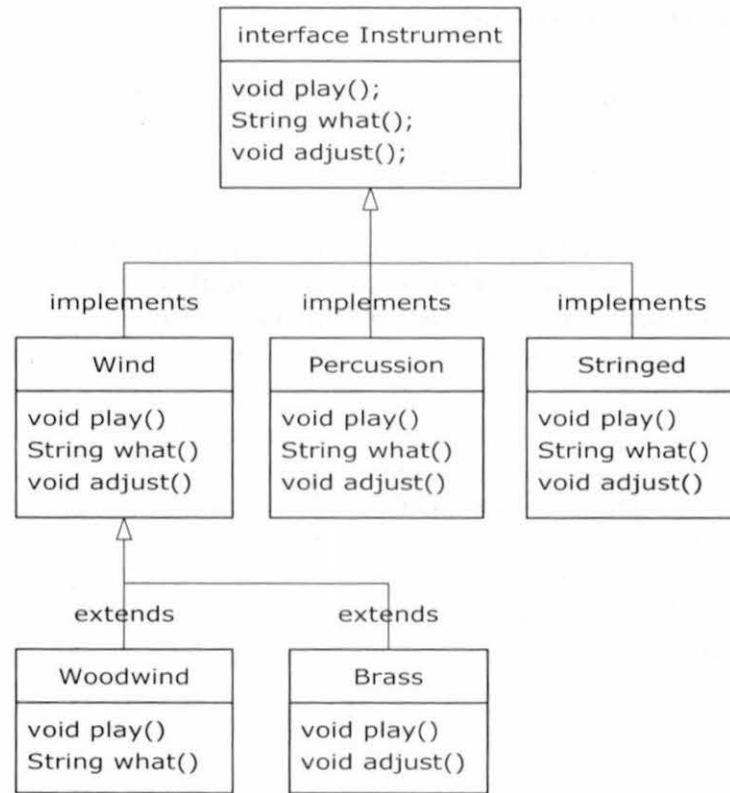
Realizzare un'interfaccia equivale a stabilire a che cosa dovranno assomigliare tutte le classi che la implementano. Pertanto, qualsiasi codice utilizzi una particolare interfaccia sa quali metodi potrebbero essere chiamati per quell'interfaccia, ma niente altro. Quindi l'interfaccia viene utilizzata per impostare un "protocollo" tra le classi, funzione che in alcuni linguaggi OOP è svolta dalla parola chiave *protocol*.

Tuttavia un'interfaccia è qualcosa di più di una semplice classe astratta portata agli estremi, poiché permette di eseguire una variazione di "ereditarietà multipla", creando una classe che può essere oggetto di upcasting a diversi tipi di base.

Per creare un'interfaccia, in luogo della parola chiave **class** si utilizza **interface**. Come nel caso delle classi è possibile anteporre la parola chiave **public**, ma soltanto se l'interfaccia è definita in un file avente lo stesso nome: in assenza dell'indicazione **public** otterrete un accesso di tipo package access, nel qual caso l'interfaccia sarà utilizzabile soltanto all'interno dello stesso pacchetto. Un'interfaccia può anche contenere campi, che tuttavia sono implicitamente **static** e **final**.

Per fare in modo che una classe sia conforme a una particolare interfaccia o gruppo di interfacce si utilizza la parola chiave **implements**: in pratica, se l'interfaccia rappresenta ciò a cui assomiglierà la classe è anche necessario definirne l'implementazione.

Oltre a questo, l'interfaccia ricorda per molti aspetti l'ereditarietà. Il diagramma relativo all'esempio degli strumenti musicali, qui di seguito, contribuisce a illustrare questi concetti.



Dalle classi **Woodwind** e **Brass** appare evidente che, una volta implementata un'interfaccia, quell'implementazione diventa una normale classe che può essere estesa come qualsiasi altra.

Potete scegliere di dichiarare in modo esplicito come **public** i metodi in un'interfaccia, che tuttavia saranno considerati **public** anche se non lo specificherete.

Al momento di realizzare l'implementazione, i metodi definiti dall'interfaccia dovranno essere dichiarati come **public**; in caso contrario i metodi avrebbero un accesso di tipo package access e questo limiterebbe l'accessibilità di un metodo durante l'ereditarietà, condizione non permessa dal compilatore Java.

Questo concetto risulta chiaro nella versione modificata dell'esempio **Instrument**. Notate che ogni metodo nell'interfaccia è rigorosamente una dichiarazione, l'unico elemento ammesso dal compilatore.

Inoltre, sebbene nessuno sia dichiarato **public**, tutti i metodi in **Instrument** sono comunque automaticamente pubblici.

```

//: interfaces/music5/Music5.java
// Interfacce.
package interfaces.music5;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

```

```

interface Instrument {
    // Costante di compilazione:
    int VALUE = 5; // static & final
    // Non puo' avere definizioni di metodo:
    void play(Note n); // Automaticamente public
    void adjust();
}

class Wind implements Instrument {
    public void play(Note n) {
        print(this + ".play() " + n);
    }
    public String toString() { return "Wind"; }
    public void adjust() { print(this + ".adjust()"); }
}

class Percussion implements Instrument {
    public void play(Note n) {
        print(this + ".play() " + n);
    }
    public String toString() { return "Percussion"; }
    public void adjust() { print(this + ".adjust()"); }
}
  
```

```

class Stringed implements Instrument {
    public void play(Note n) {
        print(this + ".play() " + n);
    }
    public String toString() { return "Stringed"; }
    public void adjust() { print(this + ".adjust()"); }
}

class Brass extends Wind {
    public String toString() { return "Brass"; }
}

class Woodwind extends Wind {
    public String toString() { return "Woodwind"; }
}

public class Music5 {
    // Non tiene conto del tipo, pertanto nuovi tipi
    // aggiunti all'applicazione funzioneranno senza problemi:
    static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }
    static void tuneAll(Instrument[] e) {
        for(Instrument i : e)
            tune(i);
    }
    public static void main(String[] args) {
        // Upcasting durante l'aggiunta all'array:
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
            new Brass(),
            new Woodwind()
        };
    }
}

```

```

        tuneAll(orchestra);
    }
} /* Output:
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
*///:~

```

Un'altra modifica in questa versione dell'esempio riguarda il metodo **what()**, che è stato cambiato in **toString()** affinché il nome rispecchiasse meglio il modo in cui veniva utilizzato il metodo. Dal momento che **toString()** è parte della classe radice **Object** non ha bisogno di essere specificato nell'interfaccia.

Il resto del codice funziona come in precedenza. Notate che non ha importanza se state procedendo all'upcasting a una classe "normale" chiamata **Instrument**, a una classe astratta **Instrument** o a un'interfaccia **Instrument**: il comportamento non cambia. In effetti, come è evidente nel metodo **tune()**, non vi è alcuna prova che **Instrument** sia una classe "normale", una classe astratta o un'interfaccia.

Esercizio 5 (2) Create un'interfaccia contenente tre metodi nel proprio **package**, e realizzate l'interfaccia in un pacchetto diverso.

Esercizio 6 (2) Dimostrate che tutti i metodi in un'interfaccia sono automaticamente **public**.

Esercizio 7 (1) Modificate l'Esercizio 9 del capitolo precedente, in modo che **Rodent** diventi un'interfaccia.

Esercizio 8 (2) In **polymorphism.Sandwich.java** (Capitolo 10) create un'interfaccia chiamata **FastFood** (con metodi appropriati) e modificate **Sandwich** in modo che implementi anche **FastFood**.

Esercizio 9 (3) Ristrutturate **Music5.java** spostando i metodi comuni di **Wind**, **Percussion** e **Stringed** in una classe astratta.

Esercizio 10 (3) Modificate **Music5.java** aggiungendo un'interfaccia **Playable** e spostate la dichiarazione **play()** da **Instrument** a **Playable**. Aggiungete **Playable** alle classi derivate, includendola nell'elenco dell'istruzione **implements**. Modificate **tune()** in modo che accetti **Playable** invece di **Instrument**.

Dissociazione completa

Se un metodo funziona con una classe invece che con un'interfaccia, siete costretti a utilizzare quella classe o le sue sottoclassi: se voleste applicare il metodo a una classe che non è inclusa in quella gerarchia, non riuscireste a farlo. Ma un'interfaccia attenua notevolmente questi vincoli, permettendovi così di riutilizzare una maggiore quantità di codice.

Per esempio, supponete di avere una classe **Processor** con i metodi **name()** e **process()** che accettano l'input, lo modificano e producono l'output. La classe di base viene estesa per creare vari tipi di **Processor**. In questo caso, i sottotipi **Processor** modificano gli oggetti **String**; osservate che i tipi di ritorno possono essere covarianti ma non tipi di argomento.

```
//: interfaces/classprocessor/Apply.java
package interfaces.classprocessor;
import java.util.*;
import static net.mindview.util.Print.*;

class Processor {
    public String name() {
        return getClass().getSimpleName();
    }
    Object process(Object input) { return input; }
}

class Upcase extends Processor {
    String process(Object input) { // ritorno covariante
        return ((String)input).toUpperCase();
    }
}

class Downcase extends Processor {
    String process(Object input) {
        return ((String)input).toLowerCase();
    }
}
```

```
class Splitter extends Processor {
    String process(Object input) {
        // L'argomento split() divide una String in frammenti:
        return Arrays.toString(((String)input).split(" "));
    }
}

public class Apply {
    public static void process(Processor p, Object s) {
        print("Using Processor " + p.name());
        print(p.process(s));
    }
    public static String s =
        "Disagreement with beliefs is by definition incorrect";
    public static void main(String[] args) {
        process(new Upcase(), s);
        process(new Downcase(), s);
        process(new Splitter(), s);
    }
} /* Output:
Using Processor Upcase
DISAGREEMENT WITH BELIEFS IS BY DEFINITION INCORRECT
Using Processor Downcase
Disagreement with beliefs is by definition incorrect
Using Processor Splitter
[Disagreement, with, beliefs, is, by, definition, incorrect]
*///:~
```

Il metodo **Apply.process()** accetta qualsiasi genere di **Processor** e lo applica a un **Object**, poi visualizza i risultati. La creazione di un metodo che si comporta diversamente in funzione dell'argomento passatogli è affidata a una struttura di progettazione nota come *Strategy design pattern*: il metodo contiene il componente fisso dell'algoritmo da eseguire e **Strategy** contiene la parte variabile. **Strategy** è l'oggetto passato, che contiene il codice da eseguire. In questo esempio l'oggetto **Processor** è **Strategy**, che in **main()** viene applicato in tre modi diversi a **String s**.



Il metodo **split()** fa parte della classe **String**: accetta l'oggetto **String**, lo suddivide utilizzando come separatore l'argomento passatogli e restituisce una **String[]**; in questo caso specifico è utilizzato come metodo abbreviato per creare un insieme di **String**.

Ora supponete di avere scoperto un insieme di filtri elettronici che potrebbero essere applicabili al vostro metodo **Apply.process()**.

```
//: interfaces/filters/Waveform.java
package interfaces.filters;

public class Waveform {
    private static long counter;
    private final long id = counter++;
    public String toString() { return "Waveform " + id; }
} ///:~

//: interfaces/filters/Filter.java
package interfaces.filters;

public class Filter {
    public String name() {
        return getClass().getSimpleName();
    }
    public Waveform process(Waveform input) { return input; }
} ///:~

//: interfaces/filters/LowPass.java
package interfaces.filters;

public class LowPass extends Filter {
    double cutoff;
    public LowPass(double cutoff) { this.cutoff = cutoff; }
    public Waveform process(Waveform input) {
        return input; // Elaborazione fittizia
    }
} ///:~
```



```
//: interfaces/filters/HighPass.java
package interfaces.filters;

public class HighPass extends Filter {
    double cutoff;
    public HighPass(double cutoff) { this.cutoff = cutoff; }
    public Waveform process(Waveform input) { return input; }
} ///:~
```

```
//: interfaces/filters/BandPass.java
package interfaces.filters;

public class BandPass extends Filter {
    double lowCutoff, highCutoff;
    public BandPass(double lowCut, double highCut) {
        lowCutoff = lowCut;
        highCutoff = highCut;
    }
    public Waveform process(Waveform input) { return input; }
} ///:~
```

Filter ha gli stessi elementi di interfaccia di **Processor**. Il creatore di questa classe non poteva sapere che volevate utilizzare **Filter** come **Processor**, quindi non ha fatto in modo che ereditasse da quest'ultimo. Di conseguenza non vi è possibile utilizzare **Filter** con il metodo **Apply.process()** anche se potrebbe benissimo funzionare. Sostanzialmente l'accoppiamento tra **Apply.process()** e **Processor** è più forte di quanto occorra e questo impedisce che il codice **Apply.process()** possa essere riutilizzato al momento opportuno. Notate anche che gli input e gli output sono entrambi di tipo **Waveform**.

Tuttavia, considerato che **Processor** è un'interfaccia, i vincoli sono abbastanza “allentati” da consentirvi il riutilizzo di un metodo **Apply.process()** che preveda quella stessa interfaccia. Queste sono le versioni modificate di **Processor** e **Apply**.

```
//: interfaces/interfaceprocessor/Processor.java
package interfaces.interfaceprocessor;
```



```

public interface Processor {
    String name();
    Object process(Object input);
} //:~

//: interfaces/interfaceprocessor/Apply.java
package interfaces.interfaceprocessor;
import static net.mindview.util.Print.*;

public class Apply {
    public static void process(Processor p, Object s) {
        print("Using Processor " + p.name());
        print(p.process(s));
    }
} //:~

```

Il primo modo per beneficiare del riutilizzo del codice è fare sì che i programmati client possano scrivere le loro classi per conformarsi all'interfaccia, come nell'esempio seguente.

```

//: interfaces/interfaceprocessor/StringProcessor.java
package interfaces.interfaceprocessor;
import java.util.*;

public abstract class StringProcessor implements Processor{
    public String name() {
        return getClass().getSimpleName();
    }
    public abstract String process(Object input);
    public static String s =
        "If she weighs the same as a duck, she's made of wood";
    public static void main(String[] args) {
        Apply.process(new Upcase(), s);
        Apply.process(new Downcase(), s);
        Apply.process(new Splitter(), s);
    }
}

```

```

class Upcase extends StringProcessor {
    public String process(Object input) { // Ritorno covariante
        return ((String)input).toUpperCase();
    }
}

class Downcase extends StringProcessor {
    public String process(Object input) {
        return ((String)input).toLowerCase();
    }
}

class Splitter extends StringProcessor {
    public String process(Object input) {
        return Arrays.toString(((String)input).split(" "));
    }
} /* Output:
Using Processor Upcase
IF SHE WEIGHS THE SAME AS A DUCK, SHE'S MADE OF WOOD
Using Processor Downcase
if she weighs the same as a duck, she's made of wood
Using Processor Splitter
[If, she, weighs, the, same, as, a, duck,, she's, made, of,
wood]
*///:~

```

Tuttavia spesso vi capiterà di non essere in grado di modificare le classi che volete utilizzare: nel caso dei filtri elettronici, per esempio, la libreria è stata “scoperta”, non creata. In questi casi potete servirvi della struttura di progettazione chiamata *Adapter design pattern*. Con *Adapter*, scrivete codice che accetta l'interfaccia corrente e produce l'interfaccia di cui avete bisogno, come nell'esempio seguente.

```

//: interfaces/interfaceprocessor/FilterProcessor.java
package interfaces.interfaceprocessor;
import interfaces.filters.*;

```

```

class FilterAdapter implements Processor {
    Filter filter;
    public FilterAdapter(Filter filter) {
        this.filter = filter;
    }
    public String name() { return filter.name(); }
    public Waveform process(Object input) {
        return filter.process((Waveform)input);
    }
}

public class FilterProcessor {
    public static void main(String[] args) {
        Waveform w = new Waveform();
        Apply.process(new FilterAdapter(new LowPass(1.0)), w);
        Apply.process(new FilterAdapter(new HighPass(2.0)), w);
        Apply.process(
            new FilterAdapter(new BandPass(3.0, 4.0)), w);
    }
} /* Output:
Using Processor LowPass
Waveform 0
Using Processor HighPass
Waveform 0
Using Processor BandPass
Waveform 0
*//*/

```

In questo approccio ad Adapter, il costruttore **FilterAdapter** accetta l'interfaccia che avete disponibile, **Filter**, e produce un oggetto che possiede l'interfaccia **Processor** che occorre. Notate anche la delega nella classe **FilterAdapter**.

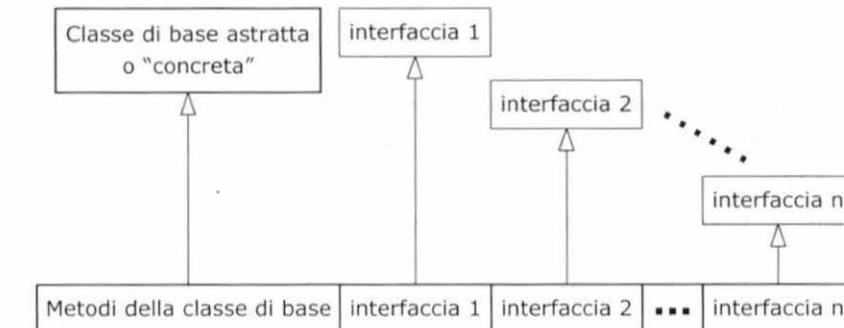
Dissociare l'interfaccia dall'implementazione permette di applicare un'interfaccia a varie implementazioni di diverso tipo e consente quindi di riutilizzare meglio il codice.

Esercizio 11 (4) Create una classe con un metodo che accetta un argomento **String** e produce un risultato nel quale ogni coppia di caratteri

dell'argomento risulta scambiata. Adattate la classe in modo che funzioni con **interfaceprocessor.Apply.process()**.

"Ereditarietà multipla" in Java

Considerato che un'interfaccia è priva di implementazione, ossia non è associata ad alcuna forma di memorizzazione, non vi è nulla che impedisca l'associazione di interfacce multiple. Questa caratteristica è veramente utile, poiché vi sono situazioni in cui è necessario affermare che "Una *x* è allo stesso tempo una *a*, una *b* e una *c*". In C++ la combinazione di diverse interfacce di classe è chiamata *ereditarietà multipla*, e talvolta può risultare onerosa poiché ogni classe può avere una sua implementazione. Java permette di ottenere lo stesso effetto, ma soltanto una classe può avere un'implementazione, pertanto non presenta i problemi di C++ nella combinazione di interfacce multiple.



In una classe derivata non è necessario che la classe di base sia **abstract** o "concreta", vale a dire priva di metodi **abstract**. Se ereditate da un oggetto che non sia un'interfaccia, dovrete ricorrere all'ereditarietà "singola" vera e propria, che ammette un solo oggetto di base: tutti gli altri elementi ottenuti per ereditarietà devono essere interfacce. La sintassi di questa tecnica prevede di indicare, dopo la parola chiave **implements**, tutti i nomi delle interfacce separati da virgolette. È possibile avere un numero illimitato di interfacce ciascuna delle quali, essendo indipendente, può essere sottoposta a upcasting. Il seguente esempio si riferisce a una classe concreta che viene combinata con numerose interfacce per produrre una nuova classe.

```
//: interfaces/Adventure.java
// Interfacce multiple.
```

```

interface CanFight {
    void fight();
}

interface CanSwim {
    void swim();
}

interface CanFly {
    void fly();
}

class ActionCharacter {
    public void fight() {}
}

class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
    public void swim() {}
    public void fly() {}
}

public class Adventure {
    public static void t(CanFight x) { x.fight(); }
    public static void u(CanSwim x) { x.swim(); }
    public static void v(CanFly x) { x.fly(); }
    public static void w(ActionCharacter x) { x.fight(); }
    public static void main(String[] args) {
        Hero h = new Hero();
        t(h); // Da gestire come CanFight
        u(h); // Da gestire come CanSwim
        v(h); // Da gestire come CanFly
        w(h); // Da gestire come ActionCharacter
    }
}

```

Esercizio 11 (difficile)
creare String e creare

Potete vedere che **Hero** combina la classe concreta **ActionCharacter** con le interfacce **CanFight**, **CanSwim** e **CanFly**. Quando combinate una classe concreta con le interfacce in questo modo, essa deve apparire per prima, seguita dalle interfacce: in caso contrario il compilatore visualizzerà un errore.

La segnatura per **fight()** è la stessa nell'interfaccia **CanFight** e nella classe **ActionCharacter**, e per **fight()** non è fornita alcuna definizione in **Hero**. Potete ampliare un'interfaccia, ma in tal caso ne otterreste una diversa. Quando volete creare un oggetto ricordate che tutte le definizioni devono già esistere. Sebbene **Hero** non fornisca esplicitamente una definizione per **fight()**, questa viene fornita con **ActionCharacter**: in questo modo è possibile creare oggetti **Hero**.

Nella classe **Adventure** sono presenti quattro metodi che accettano gli argomenti delle diverse interfacce e della classe concreta. Quando viene creato, l'oggetto **Hero** può essere passato a uno qualunque di questi metodi, il che significa che a sua volta viene sottoposto a upcasting a ogni interfaccia. Grazie alla particolare progettazione delle interfacce Java, la gestione di questi meccanismi non richiede particolare impegno da parte del programmatore.

Ricordate che una delle ragioni principali per ricorrere alle interfacce è illustrata nell'esempio precedente: la possibilità di eseguire un upcast a più di un tipo di base e la flessibilità che questo comporta.

La seconda ragione per utilizzare le interfacce, tuttavia, è la stessa che giustifica l'utilizzo di una classe di base astratta: impedire al programmatore client di produrre un oggetto di questa classe, definendo che si tratta solo di un'interfaccia.

Questo fa sorgere una domanda: è preferibile utilizzare un'interfaccia o una classe **abstract**? Quando vi è possibile creare la classe di base senza definizioni di metodo né variabili membro, dovreste privilegiare sempre le interfacce rispetto alle classi **abstract**.

In effetti, se sapete che un oggetto diventerà una classe di base potrete pensare di trasformarlo in un'interfaccia: questo argomento sarà affrontato nuovamente nel riepilogo del capitolo.

Esercizio 12 (2) In **Adventure.java** aggiungete un'interfaccia chiamata **CanClimb**, adottando lo schema delle altre interfacce.

Esercizio 13 (2) Create un'interfaccia ed ereditatene due nuove da essa. Applicate l'ereditarietà multipla creando una terza interfaccia che eredita dalla due appena create.²

2. Questo esercizio mostra come le interfacce siano in grado di impedire il verificarsi del cosiddetto "problema del diamante" (*diamond problem*), tipico dell'ereditarietà multipla in C++.

Estensione di un'interfaccia mediante ereditarietà

Ricorrendo all'ereditarietà potete aggiungere facilmente nuove dichiarazioni di metodo a un'interfaccia, nonché combinare numerose interfacce in una nuova. In entrambi i casi si ottiene una nuova interfaccia, come illustra l'esempio seguente.

```
//: interfaces/HorrorShow.java
// Estensione di un'interfaccia mediante ereditarietà.

interface Monster {
    void menace();
}

interface DangerousMonster extends Monster {
    void destroy();
}

interface Lethal {
    void kill();
}

class DragonZilla implements DangerousMonster {
    public void menace() {}
    public void destroy() {}
}

interface Vampire extends DangerousMonster, Lethal {
    void drinkBlood();
}

class VeryBadVampire implements Vampire {
    public void menace() {}
    public void destroy() {}
    public void kill() {}
    public void drinkBlood() {}
}
```

```
public class HorrorShow {
    static void u(Monster b) { b.menace(); }
    static void v(DangerousMonster d) {
        d.menace();
        d.destroy();
    }
    static void w(Lethal l) { l.kill(); }
    public static void main(String[] args) {
        DangerousMonster barney = new DragonZilla();
        u(barney);
        v(barney);
        Vampire vlad = new VeryBadVampire();
        u(vlad);
        v(vlad);
        w(vlad);
    }
} //:~
```

DangerousMonster è una semplice estensione di **Monster** che produce una nuova interfaccia, implementata in **DragonZilla**.

La sintassi utilizzata in **Vampire** funziona soltanto quando si ereditano le interfacce. Di norma potete utilizzare **extends** con una sola classe, tuttavia in fase di costruzione di una nuova interfaccia **extends** può riferirsi anche a più interfacce di base; come potete vedere, nella sintassi i nomi delle interfacce sono separati semplicemente da virgole.

Esercizio 14 (2) Create tre interfacce, ognuna con due metodi. Ereditate una nuova interfaccia che combina le tre precedenti, aggiungendo un nuovo metodo; create una classe che implementa la nuova interfaccia ed eredita anche da una classe concreta. Ora scrivete quattro metodi, ognuno dei quali accetta come argomento una delle quattro interfacce. In **main()**, create un oggetto della vostra classe e passatelo a ognuno dei metodi.

Esercizio 15 (2) Modificate l'esercizio precedente creando una classe **abstract** ed ereditandola nella classe derivata.



Collisione di nomi nella combinazione di interfacce

Quando implementate interfacce multiple potreste imbattervi in un piccolo tranello. Nell'esempio precedente, **CanFight** e **ActionCharacter** hanno entrambe un identico metodo **void fight()**. Un metodo identico non è certo un problema, ma che cosa accadrebbe se questo fosse diverso nella segnatura o nel tipo di ritorno? Ecco un esempio.

```
//: interfaces/InterfaceCollision.java
package interfaces;
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }

class C2 implements I1, I2 {
    public void f() {}
    public int f(int i) { return 1; } // sovraccarico
}
class C3 extends C implements I2 {
    public int f(int i) { return 1; } // sovraccarico
}

class C4 extends C implements I3 {
    // Identico, nessun problema:
    public int f() { return 1; }
}

// I metodi hanno tipi di ritorno diversi:
//! abstract class C5 extends C implements I1 {}
//! interface I4 extends I1, I3 {} //:/~
```

Le difficoltà si verificano poiché l'insieme di sovrascrittura (*overriding*), implementazione e sovraccarico (*overloading*) produce una miscela esplosiva; inoltre i metodi sovraccarichi non possono essere diversi soltanto per il tipo di ritorno. Se eliminate i commenti dalle ultime due righe del codice, i messaggi di errore vi diranno tutto.

*InterfaceCollision.java:24: f() in interfaces.C cannot implement
f() in interfaces.I1; attempting to use incompatible return type
found : int*

required: void

abstract class C5 extends C implements I1 {}

*InterfaceCollision.java:25: types interfaces.I3 and interfaces.I1 are
incompatible; both define f(), but with unrelated return types
interface I4 extends I1, I3 {} //:/~*

2 errors

BUILD FAILED (total time: 0 seconds)

Tra l'altro, l'utilizzo di nomi di metodo uguali in interfacce diverse che sono destinate a essere combinate genera codice confuso e poco leggibile: per questo motivo dovreste cercare di evitare questo approccio.

Come adattarsi a un'interfaccia

Uno degli aspetti più interessanti nell'utilizzo delle interfacce è la possibilità di definire più implementazioni per la stessa interfaccia. Nei casi più semplici questo si traduce in un metodo che accetta un'interfaccia, lasciando al programmatore il solo compito di implementarla e di passare l'oggetto al metodo. Pertanto, un impiego comune per le interfacce è nella struttura di progettazione *Strategy design pattern*, di cui si è già accennato. In accordo a questa struttura, scrivete un metodo che esegue determinate operazioni e che accetterà l'interfaccia che specificherete. In pratica è come se affermaste: "potete utilizzare il mio metodo con qualsiasi oggetto desideriate, purché il vostro oggetto sia conforme alla mia interfaccia". Questa tecnica rende il vostro metodo più flessibile, generico e riutilizzabile.

Per esempio il costruttore per la classe **Scanner** di Java SE5, che vedrete meglio nel Capitolo 12 dedicato alle stringhe, accetta un'interfaccia **Readable**. Scoprirete che **Readable** non è l'argomento di nessun altro metodo nella libreria standard di Java: questa interfaccia è stata creata esclusivamente per **Scanner**, in modo che **Scanner** non dovesse richiedere come argomento una classe particolare. Grazie a questo accorgimento la classe **Scanner** può funzionare con diversi tipi. Se create una nuova classe e volete che sia utilizzabile con **Scanner**, dovrete specificare l'interfaccia **Readable**, come nell'esempio seguente.

```
//: interfaces/RandomWords.java
// Implementazione di un'interfaccia conforme a un metodo.
import java.nio.*;
import java.util.*;
public class RandomWords implements Readable {
    private static Random rand = new Random(47);
```

```

private static final char[] capitals =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();
private static final char[] lowers =
    "abcdefghijklmnopqrstuvwxyz".toCharArray();
private static final char[] vowels =
    "aeiou".toCharArray();
private int count;
public RandomWords(int count) { this.count = count; }
public int read(CharBuffer cb) {
    if(count-- == 0)
        return -1; // Indica la fine dell'input
    cb.append(capitals[rand.nextInt(capitals.length)]);
    for(int i = 0; i < 4; i++) {
        cb.append(vowels[rand.nextInt(vowels.length)]);
        cb.append(lowers[rand.nextInt(lowers.length)]);
    }
    cb.append(" ");
    return 10; // Numero di caratteri accodati
}
public static void main(String[] args) {
    Scanner s = new Scanner(new RandomWords(10));
    while(s.hasNext())
        System.out.println(s.next());
}
} /* Output:
Yazeruyac
Fowenucor
Goeazimom
Raeuuacio
Nuoadesiw
Hageaikux
Ruqicibui
Numasetih
Kuuuuozog
Waqizeyoy
*///:~
```

L'interfaccia **Readable** richiede l'implementazione di un metodo **read()** al cui interno è aggiunto l'argomento **CharBuffer**, o **return -1** quando non è disponibile altro input. Tenete presente che **CharBuffer** può essere utilizzato in diversi modi: consultate la documentazione Java.

Supponete di avere una classe in cui **Readable** non sia già implementata: come fare per renderla funzionante con **Scanner**? Considerate l'esempio di una classe che produce numeri in virgola mobile casuali.

```

//: interfaces/RandomDoubles.java
import java.util.*;

public class RandomDoubles {
    private static Random rand = new Random(47);
    public double next() { return rand.nextDouble(); }
    public static void main(String[] args) {
        RandomDoubles rd = new RandomDoubles();
        for(int i = 0; i < 7; i++)
            System.out.print(rd.next() + " ");
    }
} /* Output:
0.7271157860730044 0.5309454508634242 0.16020656493302599
0.1884786697771732 0.5166020801268457 0.2678662084200585
0.2613610344283964
*///:~
```

Anche adesso potete servirvi della struttura *Adapter design pattern*, ma in questo caso specifico la classe adattata può essere creata per ereditarietà e implementando l'interfaccia **Readable**.

Quindi, servendovi della funzionalità di “pseudo-ereditarietà multipla” offerta dalla parola chiave **interface**, potete produrre una nuova classe che sia simultaneamente **RandomDoubles** e **Readable**.

```

//: interfaces/AdaptedRandomDoubles.java
// Crea un adapter per ereditarieta'.
import java.nio.*;
import java.util.*;
```

```

public class AdaptedRandomDoubles extends RandomDoubles
implements Readable {
    private int count;
    public AdaptedRandomDoubles(int count) {
        this.count = count;
    }
    public int read(CharBuffer cb) {
        if(count-- == 0)
            return -1;
        String result = Double.toString(next()) + " ";
        cb.append(result);
        return result.length();
    }
    public static void main(String[] args) {
        Scanner s = new Scanner(new AdaptedRandomDoubles(7));
        while(s.hasNextDouble())
            System.out.print(s.nextDouble() + " ");
    }
} /* Output:
0.7271157860730044 0.5309454508634242 0.16020656493302599
0.1884786697771732 0.5166020801268457 0.2678662084200585
0.2613610344283964
*///:~

```

Dal momento che questo consente di aggiungere un'interfaccia a qualsiasi classe esistente, significa che un metodo che accetta un'interfaccia costituisce una valida tecnica per rendere funzionante qualsiasi classe con quel metodo: questa, in sintesi, è la potenza offerta dall'utilizzo delle interfacce rispetto alle classi.

Esercizio 16 (3) Create una classe che produce una sequenza di **char** e adattatela in modo che diventi un input per l'oggetto **Scanner**.

Campi nelle interfacce

Poiché qualsiasi campo incluso in un'interfaccia è automaticamente **static** e **final**, l'interfaccia rappresenta un eccellente strumento per creare gruppi di valori costanti; prima di Java SE5, infatti, questo era l'unico modo per produrre lo stesso effetto di una **enum** di C o C++.

Per questo motivo potrete trovare codice precedente a Java SE5 strutturato nel modo seguente.

```

//: interfaces/Months.java
// Utilizzo delle interfacce per la creazione di gruppi di
// costanti.
package interfaces;

public interface Months {
    int
        JANUARY = 1, FEBRUARY = 2, MARCH = 3,
        APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
        AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
        NOVEMBER = 11, DECEMBER = 12;
} //://:~

```

Noteate il tipico stile Java che si serve delle sole lettere maiuscole, con sottolineature per separare le parole multiple di un singolo identificativo, per valori **static final** caratterizzati da inizializzatori costanti. I campi in un'interfaccia sono automaticamente **public**, pertanto questo non viene specificato in modo esplicito.

Java SE5 ha reso disponibile la parola chiave **enum**, molto più potente e flessibile, pertanto per le costanti non ha più senso utilizzare le interfacce. Tuttavia, consultando il codice di vecchie applicazioni potrete incontrare la forma antiquata: i supplementi a questo manuale, disponibili all'indirizzo www.mindview.net, contengono una descrizione completa dell'approccio precedente a Java SE5 per la produzione di tipi enumerativi mediante interfacce. Troverete maggiori dettagli sull'uso di **enum** nel Volume 2, Capitolo 7 dedicato ai tipi enumerativi.

Esercizio 17 (2) Dimostrate che i campi in un'interfaccia sono implicitamente **static** e **final**.

Inizializzare i campi nelle interfacce

I campi definiti nelle interfacce non possono essere di tipo “blank **final**”, ma possono essere inizializzati con espressioni non costanti.

```

//: interfaces/RandVals.java
// Inizializzazione di campi di interfaccia
// con inizializzatori non costanti.
import java.util.*;

```



```
public interface RandVals {
    Random RAND = new Random(47);
    int RANDOM_INT = RAND.nextInt(10);
    long RANDOM_LONG = RAND.nextLong() * 10;
    float RANDOM_FLOAT = RAND.nextLong() * 10;
    double RANDOM_DOUBLE = RAND.nextDouble() * 10;
} //:~
```

Essendo **static** i campi vengono inizializzati al primo caricamento della classe, che avviene quando si accede per la prima volta a uno qualsiasi di essi. Ecco un semplice test.

```
//: interfaces/TestRandVals.java
import static net.mindview.util.Print.*;

public class TestRandVals {
    public static void main(String[] args) {
        print(RandVals.RANDOM_INT);
        print(RandVals.RANDOM_LONG);
        print(RandVals.RANDOM_FLOAT);
        print(RandVals.RANDOM_DOUBLE);
    }
} /* Output:
8
-32032247016559954
-8.5939291E18
5.779976127815049
*//:~
```

Naturalmente i campi non fanno parte dell'interfaccia e i valori vengono memorizzati nell'area di memoria statica riservata a essa.

Interfacce nidificate

Le interfacce possono essere nidificate all'interno di classi e di altre interfacce.³

3. L'autore ringrazia Martin Danner per avere proposto l'argomento durante un seminario.



Questa funzionalità offre un certo numero di caratteristiche interessanti:

```
//: interfaces/nesting/NestingInterfaces.java
package interfaces.nesting;

class A {
    interface B {
        void f();
    }
    public class BImp implements B {
        public void f() {}
    }
    private class BImp2 implements B {
        public void f() {}
    }
    public interface C {
        void f();
    }
    class CImp implements C {
        public void f() {}
    }
    private class CImp2 implements C {
        public void f() {}
    }
    private interface D {
        void f();
    }
    private class DImp implements D {
        public void f() {}
    }
    public class DImp2 implements D {
        public void f() {}
    }
    public D getD() { return new DImp2(); }
    private D dRef;
    public void receiveD(D d) {
        dRef = d;
```



```

dRef.f();
}

}

interface E {
    interface G {
        void f();
    }
    // "public" ridondante:
    public interface H {
        void f();
    }
    void g();
    // Non puo' essere private nell'ambito di un'interfaccia:
    //! private interface I {}
}

public class NestingInterfaces {
    public class BImp implements A.B {
        public void f() {}
    }
    class CImp implements A.C {
        public void f() {}
    }
    // E' possibile implementare un'interfaccia private soltanto
    // nell'ambito della classe di definizione di
    // quell'interfaccia:
    //! class DImp implements A.D {
    //!     public void f() {}
    //! }
    class EImp implements E {
        public void g() {}
    }
    class EGImp implements E.G {
        public void f() {}
    }
}

```

Interfacce

Le interfacce

```

class EIimp2 implements E {
    public void g() {}
    class EG implements E.G {
        public void f() {}
    }
}
public static void main(String[] args) {
    A a = new A();
    // Impossibile accedere a A.D:
    //! A.D ad = a.getD();
    // Restituisce soltanto A.D:
    //! A.DImp2 di2 = a.getD();
    // Non puo' accedere a un membro dell'interfaccia:
    //! a.getD().f();
    // Soltanto un'altra A puo' fare qualsiasi cosa con
    // getD():
    A a2 = new A();
    a2.receiveD(a.getD());
}
} //:~

```

La sintassi per nidificare un'interfaccia all'interno di una classe è ragionevolmente ovvia. Come per le interfacce non nidificate, la visibilità può essere di tipo **public** o **package access**.

Le interfacce possono anche essere **private**, come potete notare in **A.D**, e la stessa sintassi di qualificazione è utilizzata sia per le interfacce nidificate sia per le classi nidificate. Qual è lo scopo di un'interfaccia nidificata **private**? Potreste ritenere che sia implementabile unicamente come classe interna **private**, come in **DImp**, ma **A.DImp2** mostra che può essere implementata anche come classe **public**.

Tuttavia, **A.DImp2** può essere utilizzato soltanto così com'è. Non è permesso fare riferimento al fatto che **A.DImp2** implementa l'interfaccia **private D**, pertanto l'implementazione di un'interfaccia **private** è un modo per forzare la definizione dei metodi in quell'interfaccia senza aggiungere informazioni sul tipo, vale a dire senza consentire l'upcasting.

Il metodo **getD()** introduce un'ulteriore difficoltà nell'interfaccia **private**: si tratta di un metodo **public** che restituisce un riferimento a un'interfaccia **private**. Che cosa è possibile fare con il valore restituito da questo metodo? In **main()**



potete vedere numerosi tentativi per utilizzare il valore di ritorno, tutti infruttuosi. L'unico tentativo che va a buon fine si verifica quando il valore restituito viene passato a un oggetto che ha l'autorizzazione a utilizzarlo: in questo caso specifico, un'altra A, per mezzo del metodo `receiveD()`.

L'interfaccia E dimostra che le interfacce possono essere nidificate all'interno di altre. Tuttavia i vincoli cui sono soggette, soprattutto il fatto che tutti gli elementi dell'interfaccia devono essere **public**, in questo caso sono applicati in modo rigoroso, cosicché un'interfaccia nidificata in un'altra è automaticamente **public** e non può essere resa **private**.

NestingInterfaces illustra i diversi modi in cui è possibile implementare le interfacce nidificate. In particolare notate che quando implementate un'interfaccia Java non richiede di implementare un'interfaccia nidificata al suo interno. Inoltre le interfacce **private** non sono implementabili fuori dalle loro classi di definizione.

Potreste pensare che queste caratteristiche siano state introdotte soltanto per motivi di compatibilità sintattica, tuttavia è opinione generale che una volta preso atto della sua disponibilità non è difficile trovare situazioni in cui una funzionalità potrebbe rivelarsi utile.

Interfacce e factory

Un'interfaccia è concepita per essere la porta d'accesso a più implementazioni e una tecnica sfruttata comunemente per produrre oggetti adattabili a essa è la struttura di progettazione cosiddetta *Factory Method design pattern*. Invece di chiamare direttamente un costruttore, chiamate un metodo di creazione su un oggetto-factory che produce un'implementazione dell'interfaccia: teoricamente, così facendo il vostro codice risulta isolato dall'implementazione dell'interfaccia, rendendovi possibile il passaggio da un'implementazione a un'altra in modo del tutto trasparente. Di seguito è presentato un esempio della struttura *Factory Method*.

```
//: interfaces/Factories.java
import static net.mindview.util.Print.*;

interface Service {
    void method1();
    void method2();
}

class Implementation1 implements Service {
    public void method1() {print("Implementation1 method1");}
    public void method2() {print("Implementation1 method2");}
}

class Implementation2 implements Service {
    public void method1() {print("Implementation2 method1");}
    public void method2() {print("Implementation2 method2");}
}
```



```
interface ServiceFactory {
    Service getService();
}

class Implementation1Factory implements ServiceFactory {
    Implementation1() {} // Accesso di tipo package access
    public void method1() {print("Implementation1 method1");}
    public void method2() {print("Implementation1 method2");}
}

class Implementation2Factory implements ServiceFactory {
    Implementation2() {} // Accesso di tipo package access
    public void method1() {print("Implementation2 method1");}
    public void method2() {print("Implementation2 method2");}
}

public class Factories {
    public static void serviceConsumer(ServiceFactory fact) {
        Service s = fact.getService();
        s.method1();
        s.method2();
    }
    public static void main(String[] args) {
        serviceConsumer(new Implementation1Factory());
        // Le implementazioni sono completamente intercambiabili:
    }
}
```



```

    serviceConsumer(new Implementation2Factory());
}
} /* Output:
Implementation1 method1
Implementation1 method2
Implementation2 method1
Implementation2 method2
*///:~

```

Senza ricorrere al Factory Method, in qualche punto del codice dovreste specificare il tipo esatto di **Service** in costruzione, per chiamare il costruttore appropriato.

Perché aggiungere questo livello supplementare di *indirezione*? Una ragione comune è la creazione di una struttura. Supponete di creare un'applicazione che consenta, per esempio, di giocare a dama e a scacchi sulla stessa scacchiera.

```

//: interfaces/Games.java
// Un framework di gioco che utilizza i Factory Method.
import static net.mindview.util.Print.*;

interface Game { boolean move(); }
interface GameFactory { Game getGame(); }

class Checkers implements Game {
    private int moves = 0;
    private static final int MOVES = 3;
    public boolean move() {
        print("Checkers move " + moves);
        return ++moves != MOVES;
    }
}

class CheckersFactory implements GameFactory {
    public Game getGame() { return new Checkers(); }
} void net.mindview.util.Print.*;

```



```

class Chess implements Game {
    private int moves = 0;
    private static final int MOVES = 4;
    public boolean move() {
        print("Chess move " + moves);
        return ++moves != MOVES;
    }
}

class ChessFactory implements GameFactory {
    public Game getGame() { return new Chess(); }
}

public class Games {
    public static void playGame(GameFactory factory) {
        Game s = factory.getGame();
        while(s.move())
            ;
    }
    public static void main(String[] args) {
        playGame(new CheckersFactory());
        playGame(new ChessFactory());
    }
} /* Output:
Checkers move 0
Checkers move 1
Checkers move 2
Chess move 0
Chess move 1
Chess move 2
Chess move 3
*///:~

```

Se la classe **Games** fosse una porzione di codice particolarmente complessa, questo approccio permetterebbe di riutilizzare lo stesso codice con vari tipi di giochi: potete immaginare giochi ben più elaborati che trarrebbero vantaggio da questa struttura.



Nel prossimo capitolo vedrete un modo di implementazione delle factory più elegante, che ricorre alle classi interne anonime.

Esercizio 18 (2) Create un'interfaccia **Cycle** con le implementazioni **Unicycle**, **Bicycle** e **Tricycle**; generate una factory per ogni tipo di **Cycle** e il codice che le utilizza.

Esercizio 19 (3) Utilizzando Factory Method create una struttura che esegua sia il lancio di una moneta sia quello dei dadi.

Riepilogo

Forse è presuntuoso affermare che, poiché le interfacce sono uno strumento veramente valido, dovreste sempre scegliere di adottarle in luogo delle classi concrete. Naturalmente, quasi in ogni occasione, quando create una classe potreste invece creare un'interfaccia e una factory. Molti programmati sono caduti in questa tentazione, creando interfacce e factory ovunque fosse possibile. La logica sembra indicare che, tenuto conto del fatto che potreste avere bisogno di utilizzare una diversa implementazione, dovreste sempre includere questo livello di astrazione. È diventata una sorta di ottimizzazione progettuale prematura.

Qualsiasi astrazione dovrebbe essere motivata da necessità reali: le interfacce dovrebbero essere qualcosa che sia possibile ristrutturare quando necessario, non un livello supplementare di indirezione (e di complessità) da introdurre a tutti i costi nei vostri progetti. Si tratta di una complessità notevole: se vi capitasse di riscontrarla nel codice di un altro programmatore e di rendervi conto che una simile complessità è stata introdotta senza alcuna giustificazione reale (ma solo “nel caso che”), di certo mettereste in discussione tutti i progetti ai quali questa persona ha partecipato.

Un'indicazione appropriata è preferire le classi alle interfacce. Iniziate con classi: se vi renderete conto che le interfacce sono indispensabili, ristrutturate il progetto. Le interfacce sono uno strumento eccellente, di cui però è facile abusare.

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

Capitolo 10

Le classi interne



Java consente di inserire all'interno di una definizione di classe la definizione di un'altra: quest'ultima viene chiamata *classe interna* o *inner class*.

La classe interna è una funzionalità utile in quanto permette di raggruppare le classi secondo criteri logici e di controllare la visibilità di una classe nell'ambito di un'altra. Tuttavia, è importante capire che concettualmente le classi interne sono diverse dal meccanismo della composizione.

A prima vista le classi interne rammentano un semplice meccanismo di occultamento del codice: l'inserimento di classi in altre classi. Scoprirete, tuttavia, che la classe interna è in grado di offrire molto più di questo, poiché conosce e può comunicare con quella che la circonda (*classe esterna*); vedrete anche che grazie alle classi interne è possibile scrivere codice più elegante e chiaro, sebbene ciò non sia evidentemente garantito.

Inizialmente le classi interne possono apparire strane e avrete bisogno di tempo prima di potervene servire con disinvoltura nei vostri progetti. La necessità di usare classi interne non è sempre evidente: dopo la descrizione della sintassi e della semantica di base, tuttavia, il paragrafo “Perché utilizzare le classi interne?” dovrebbe chiarire i vantaggi del loro utilizzo.

Il resto del capitolo è dedicato a una dettagliata analisi della sintassi delle classi interne: queste nozioni vengono fornite soltanto per completezza del linguaggio poiché, se non altro al principio, probabilmente non avrete bisogno di servirvene.

La parte iniziale di questo capitolo, quindi, potrebbe essere tutto ciò di cui avete bisogno, almeno per il momento; se lo desiderate potete tralasciare i paragrafi più tecnici e leggerli in seguito come documentazione di riferimento.

Creazione delle classi interne

La creazione di una classe interna avviene esattamente come si è detto, inserendo la definizione di una classe all'interno di un'altra.

```
//: innerclasses/Parcel1.java
// Creazione di classi interne.

public class Parcel1 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    // Dentro Parcel1, l'utilizzo delle classi interne
    // ricorda quello di qualsiasi altra classe:
    public void ship(String dest) {
        Contents c = new Contents();
        Destination d = new Destination(dest);
        System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
        Parcel1 p = new Parcel1();
```

```
p.ship("Tasmania");
}
} /* Output:
Tasmania
*///:~
```

Le classi interne utilizzate nel metodo `ship()` sono del tutto analoghe a quelle normali: la differenza pratica è che i nomi sono nidificati all'interno della classe **Parcel1**. Vedrete tra breve che questa non è però l'unica differenza effettiva.

Tipicamente una classe esterna ha un metodo che restituisce un riferimento a una classe interna, come potete vedere nei metodi `to()` e `contents()`.

```
//: innerclasses/Parcel2.java
// Restituzione di un riferimento a una classe interna.

public class Parcel2 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    public Destination to(String s) {
        return new Destination(s);
    }
    public Contents contents() {
        return new Contents();
    }
    public void ship(String dest) {
        Contents c = contents();
        Destination d = to(dest);
```



```

        System.out.println(d.readLabel());
    }

    public static void main(String[] args) {
        Parcel12 p = new Parcel12();
        p.ship("Tasmania");
        Parcel12 q = new Parcel12();
        // Definizione dei riferimenti alle classi interne:
        Parcel12.Contents c = q.contents();
        Parcel12.Destination d = q.to("Borneo");
    }
} /* Output:
Tasmania
*//*:~
```

Come potete notare in **main()**, se volete creare un oggetto di una classe interna in un punto qualsiasi che non sia all'interno di un metodo non **static** della classe esterna, dovrete specificare il tipo dell'oggetto come **NomeClasseEsterna.NomeClasseInterna**.

Esercizio 1 (1) Scrivete una classe chiamata **Outer** che contiene una classe interna chiamata **Inner**. Aggiungete a **Outer** un metodo che restituisce un oggetto di tipo **Inner**; in **main()**, infine, create e inizializzate un riferimento a un oggetto **Inner**.

Collegamento alla classe esterna

Finora sembra che le classi interne non siano altro che una tecnica di occultamento dei nomi e uno schema di organizzazione del codice, funzionalità certamente utili ma non irrinunciabili.

Ma c'è dell'altro: quando create una classe interna, ogni suo oggetto possiede un collegamento all'oggetto "inclusivo" (esterno) che l'ha creato, e può quindi accedere ai membri dell'oggetto inclusivo senza particolari requisiti. Inoltre, le classi interne hanno diritti di accesso su tutti gli elementi della classe inclusiva.¹

1. Si tratta di un meccanismo profondamente diverso da quello delle classi nidificate di C++, che è soltanto una tecnica per l'occultamento dei nomi: C++ non mette a disposizione alcun collegamento all'oggetto esterno, né diritti di accesso impliciti.



L'esempio seguente illustra questo comportamento.

```

//: innerclasses/Sequence.java
// Contiene una sequenza di Objects.

interface Selector {
    boolean end();
    Object current();
    void next();
}

public class Sequence {
    private Object[] items;
    private int next = 0;
    public Sequence(int size) { items = new Object[size]; }
    public void add(Object x) {
        if(next < items.length)
            items[next++] = x;
    }
    private class SequenceSelector implements Selector {
        private int i = 0;
        public boolean end() { return i == items.length; }
        public Object current() { return items[i]; }
        public void next() { if(i < items.length) i++; }
    }
    public Selector selector() {
        return new SequenceSelector();
    }
}

public static void main(String[] args) {
    Sequence sequence = new Sequence(10);
    for(int i = 0; i < 10; i++)
        sequence.add(Integer.toString(i));
    Selector selector = sequence.selector();
    while(!selector.end()) {
```



```

        System.out.print(selector.current() + " ");
        selector.next();
    }
}
/* Output:
0 1 2 3 4 5 6 7 8 9
*//*:~
```

Sequence è semplicemente un array a dimensione fissa di **Object**, inglobato in una classe. Chiamando il metodo **add()** potete aggiungere un nuovo **Object** alla fine della sequenza, se vi è spazio disponibile. Per ottenere ciascuno degli oggetti contenuti in una **Sequence** è disponibile un'interfaccia chiamata **Selector**: questo è un esempio della struttura *Iterator design pattern*, che verrà trattata in modo più approfondito nel prosieguo del manuale. Per mezzo di un **Selector** potete verificare se vi trovate al termine della sequenza (**end()**), accedere all'**Object** corrente (**current()**) e passare all'**Object** successivo (**next()**) contenuto nell'oggetto **Sequence**. Poiché **Selector** è un'interfaccia, altre classi possono implementarla a loro piacimento e altri metodi possono accettare questa interfaccia come argomento per creare codice di utilizzo più comune.

In questo esempio **SequenceSelector** è una classe **private** che fornisce le funzionalità di **Selector**; in **main()** potete osservare la creazione di una **Sequence**, seguita dall'aggiunta di alcuni oggetti **String**. Quindi viene generato un **Selector** tramite una chiamata al metodo **selector()**; l'oggetto risultante è utilizzato per spostarsi attraverso la sequenza **Sequence** e selezionare ogni elemento.

A prima vista la creazione di un oggetto **SequenceSelector** assomiglia a una comune classe interna, ma un'analisi più attenta indica che i metodi **end()**, **current()** e **next()** fanno riferimento a **items**, elemento che non fa parte di **SequenceSelector** ma è un campo **private** della classe inclusiva. Tuttavia la classe interna può accedere ai metodi e ai campi della classe inclusiva come se ne fosse proprietaria. Questo procedimento è molto pratico, come potete vedere nell'esempio precedente.

Alla classe interna è quindi garantito l'accesso automatico ai membri della classe esterna, ma come? La classe interna intercetta segretamente un riferimento allo specifico oggetto della classe inclusiva che è responsabile della sua creazione. Pertanto, quando fate riferimento a un membro della classe inclusiva, esso viene utilizzato per identificare tale membro. Fortunatamente è il compilatore a occuparsi di questi dettagli, ma è necessario sapere che un oggetto di una classe interna può essere creato soltanto in combinazione con



un oggetto della classe esterna (quando, come vedrete, quest'ultima non è **static**). La costruzione dell'oggetto della classe interna richiede il riferimento all'oggetto della classe inclusiva; se il compilatore non è in grado di accedere a tale riferimento visualizzerà messaggi opportuni. Di norma la costruzione dell'oggetto ha luogo senza alcun intervento da parte del programmatore.

Esercizio 2 (1) Create una classe contenente un oggetto **String** e dotata di un metodo **toString()** che visualizza questa **String**. Aggiungete alcune istanze della vostra nuova classe a un oggetto **Sequence**, quindi visualizzatele.

Esercizio 3 (1) Modificate l'Esercizio 1 in modo che **Outer** possieda un campo **private String** inizializzato dal costruttore e **Inner** abbia un metodo **toString()** che visualizza questo campo. Create un oggetto di tipo **Inner** e visualizzate il campo.

Utilizzo di **.this** e **.new**

Per generare il riferimento all'oggetto della classe esterna si indica il nome della classe seguito da un punto (.) e dalla parola chiave **this**: il riferimento risultante è automaticamente il tipo corretto, che essendo già noto e verificato al momento della compilazione non produce alcun onere elaborativo aggiuntivo in fase di esecuzione. L'esempio seguente mostra come usare **.this**.

```

//: innerclasses/DotThis.java
// "Accesso abilitativo" (qualifying access) all'oggetto della
// classe esterna.

public class DotThis {
    void f() { System.out.println("DotThis.f()"); }
    public class Inner {
        public DotThis outer() {
            return DotThis.this;
            // Un semplice "this" indicherebbe Inner
        }
        public Inner inner() { return new Inner(); }
        public static void main(String[] args) {
            DotThis dt = new DotThis();
```



```

DotThis.Inner dti = dt.inner();
dti.outer().f();
}
} /* Output:
DotThis.f()
*///:~

```

Nulla vieta di ordinare a un oggetto di creare un oggetto di una delle sue classi interne. A questo scopo dovete fornire un riferimento dell'oggetto della classe esterna in un'espressione **new**, applicando la sintassi **.new** mostrata di seguito.

```

//: innerclasses/DotNew.java
// Creazione diretta di una classe interna applicando la
// sintassi .new.

public class DotNew {
    public class Inner {}
    public static void main(String[] args) {
        DotNew dn = new DotNew();
        DotNew.Inner dni = dn.new Inner();
    }
} ///:~

```

Contrariamente a quanto potreste supporre, per creare direttamente un oggetto della classe interna non vi servirete della stessa tecnica che fa riferimento al nome della classe esterna **DotNew**: dovete invece usare un oggetto della classe esterna per creare un oggetto della classe interna, come vedete nell'esempio. Questo accorgimento risolve anche il problema di ambito di nomi per la classe interna, pertanto non dovete indicare **dn.new DotNew.Inner()**.

Non è possibile creare un oggetto della classe interna a meno di non disporre già di un oggetto della classe esterna; questa limitazione è dovuta al fatto che l'oggetto della classe interna è collegato implicitamente all'oggetto della classe esterna da cui è stato ottenuto. Tuttavia, se create una classe nidificata, nota anche come *nested class* o *classe interna static*, non avrete bisogno di un riferimento all'oggetto esterno.

Il compilatore...
- partito di una classe...



L'esempio che segue si riferisce all'utilizzo di **.new** applicato all'esempio "Parcel":

```

//: innerclasses/Parcel3.java
// Utilizzo di .new per creare istanze di una classe interna.

public class Parcel3 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) { label = whereTo; }
        String readLabel() { return label; }
    }
    public static void main(String[] args) {
        Parcel3 p = new Parcel3();
        // Per creare un'istanza della classe interna
        // si deve usare l'istanza della classe esterna:
        Parcel3.Contents c = p.new Contents();
        Parcel3.Destination d = p.new Destination("Tasmania");
    }
} ///:~

```

Esercizio 4 (2) Aggiungete un metodo alla classe **Sequence.Sequence-Selector** che genera il riferimento alla classe esterna **Sequence**.

Esercizio 5 (1) Create una classe con una classe interna e in un'altra classe create un'istanza della classe interna.

Classi interne e upcasting

Le classi interne divengono realmente "vostre" quando ne eseguite l'upcasting a una classe di base, in particolare a un'interfaccia (tenete presente che l'effetto derivato dalla creazione di un riferimento a un'interfaccia da parte di un oggetto che la implementa è essenzialmente identico all'upcasting a una classe di base); la classe interna, ovvero l'implementazione dell'inter-

faccia, può essere resa invisibile e non disponibile, un accorgimento utile per nascondere l'implementazione: tutto ciò che ottenete è un riferimento alla classe di base o all'interfaccia.

Potete quindi creare interfacce per gli esempi precedenti.

```
//: innerclasses/Destination.java
public interface Destination {
    String readLabel();
} //:~

//: innerclasses/Contents.java
public interface Contents {
    int value();
} //:~
```

Ora **Contents** e **Destination** diventano interfacce disponibili al programmatore client. Ricordate che un'interfaccia rende automaticamente **public** tutti suoi membri.

Quando ottenete un riferimento alla classe di base o all'interfaccia, è possibile che non possiate neppure scoprire il tipo esatto, come dimostra l'esempio seguente.

```
//: innerclasses/TestParcel.java

class Parcel4 {
    private class PContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    protected class PDestination implements Destination {
        private String label;
        private PDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() { return label; }
    }
}
```

```
public Destination destination(String s) {
    return new PDestination(s);
}
public Contents contents() {
    return new PContents();
}
}

public class TestParcel {
    public static void main(String[] args) {
        Parcel4 p = new Parcel4();
        Contents c = p.contents();
        Destination d = p.destination("Tasmania");
        // Illegale -- impossibile accedere alla classe private
        //!! Parcel4.PContents pc = p.new PContents();
    }
} //:~
```

Nella classe **Parcel4** è stata inclusa la nuova classe interna **PContents**, che è **private** affinché soltanto **Parcel4** possa accedervi. Le classi normali, cioè non interne, non possono essere rese **private** o **protected**: gli unici tipi di accesso consentiti sono **public** e package access. **PDestination** è **protected**, così soltanto **Parcel4**, le classi nello stesso package (ricordate che **protected** fornisce anche l'accesso di pacchetto), e chi eredita da **Parcel4** possono accedere a **PDestination**: questo significa che il programmatore client ha una conoscenza e un accesso limitati su questi membri. Infatti non è neppure possibile eseguire il downcast a una classe interna **private**, o a una **protected** che non sia stata ereditata, poiché non si ha accesso al nome: questa condizione è illustrata nella classe **TestParcel**. Così, la classe interna **private** offre al progettista delle classi un modo per evitare qualsiasi dipendenza dal codice tipizzato e per nascondere tutti i dettagli dell'implementazione. Tenete presente anche che l'estensione di un'interfaccia non procura alcun beneficio al programmatore client, il quale non può accedere a metodi aggiuntivi che non siano parte dell'interfaccia **public**. Questo consente anche al compilatore Java di generare codice più efficiente.

Esercizio 6 (2) Create un'interfaccia con almeno un metodo, in un suo package, quindi generate una classe in un package separato e aggiungete una classe interna **protected** che implementa l'interfaccia. In un terzo package create una classe per ereditarietà dalla precedente e, in



un metodo, restituite un oggetto della classe interna **protected**, eseguendo un upcasting all'interfaccia durante l'operazione di restituzione dell'oggetto.

Esercizio 7 (2) Create una classe con un campo e un metodo entrambi **private**. Generate una classe interna con un metodo che modifica il campo della classe esterna e chiama il metodo della classe esterna. In una seconda classe esterna create un oggetto della classe interna e chiamate il suo metodo, quindi mostratene l'effetto sull'oggetto della classe esterna.

Esercizio 8 (2) Determinate se una classe esterna può accedere agli elementi **private** della sua classe interna.

Classi interne, metodi e ambiti

Finora avete visto l'impiego tipico delle classi interne: di solito il codice che scriverete e leggerete conterrà classi interne "chiare", semplici e facili da capire. Ma la sintassi delle classi interne mette a vostra disposizione molte altre tecniche, ben più "oscure".

Le classi interne possono essere create all'interno di un metodo o anche in un ambito arbitrario. Sono due i motivi che potrebbero indurvi a procedere in questo senso.

1. Come avete visto in precedenza, l'implementazione di un'interfaccia di qualsiasi tipo che vi permetta di creare e restituire un riferimento.
2. La risoluzione di un problema complesso che richieda la creazione di una classe di supporto, che tuttavia non volete rendere disponibile.

Nei prossimi esempi il codice precedente verrà modificato al fine di utilizzarne:

1. una classe definita all'interno di un metodo;
2. una classe definita in un ambito interno a un metodo;
3. una classe anonima che implementa un'interfaccia;
4. una classe anonima che estende una classe dotata di un costruttore non predefinito;
5. una classe anonima che esegue l'inizializzazione dei campi;
6. una classe anonima che viene costruita mediante inizializzazione dell'istanza; le classi interne anonime non possono avere costruttori.



Il primo esempio presenta la creazione della cosiddetta *classe interna locale* (*local inner class*), vale a dire una classe nell'ambito di un metodo, invece che nell'ambito di un'altra classe.

```
//: innerclasses/Parcel5.java
// Come nidificare una classe in un metodo.

public class Parcel5 {
    public Destination destination(String s) {
        class PDestination implements Destination {
            private String label;
            private PDestination(String whereTo) {
                label = whereTo;
            }
            public String readLabel() { return label; }
        }
        return new PDestination(s);
    }
    public static void main(String[] args) {
        Parcel5 p = new Parcel5();
        Destination d = p.destination("Tasmania");
    }
} //:~
```

La classe **PDestination** fa parte del metodo **destination()** invece che di **Parcel5**, di conseguenza non è possibile accedere a **PDestination** al di fuori di **destination()**.

Notate l'upcasting che si verifica nell'istruzione **return**: niente esce da **destination()** tranne un riferimento a **Destination**, la classe di base. Naturalmente, il fatto che il nome della classe **PDestination** sia posto all'interno di **destination()** non significa affatto che **PDestination** non sia un oggetto valido, una volta che **destination()** restituirà un valore.

Potreste utilizzare l'identificativo della classe **PDestination** per una classe interna dentro ogni classe nella stessa sottodirectory, senza generare conflitti di nome.



L'esempio successivo mostra come nidificare una classe interna all'interno di un ambito qualsiasi.

```
//: innerclasses/Parcel6.java
// Nidificazione di una classe in un ambito.

public class Parcel6 {
    private void internalTracking(boolean b) {
        if(b) {
            class TrackingSlip {
                private String id;
                TrackingSlip(String s) {
                    id = s;
                }
                String getSlip() { return id; }
            }
            TrackingSlip ts = new TrackingSlip("slip");
            String s = ts.getSlip();
        }
        // Non e' possibile farlo qui perche' e' fuori ambito:
        //! TrackingSlip ts = new TrackingSlip("x");
    }
    public void track() { internalTracking(true); }
    public static void main(String[] args) {
        Parcel6 p = new Parcel6();
        p.track();
    }
} ///:~
```

La classe **TrackingSlip** è nidificata nell'ambito di un'istruzione **if**, senza che per questo la sua creazione sia soggetta a condizioni: viene compilata insieme al resto del codice, semplicemente non è disponibile fuori dell'ambito in cui è definita; se si trascura questo dettaglio, è esattamente come una classe comune.

Esercizio 9 (1) Create un'interfaccia con almeno un metodo e implementatela definendo una classe interna a un metodo, che restituisce un riferimento alla vostra interfaccia.



Esercizio 10 (1) Ripetete l'esercizio precedente, definendo però la classe interna in un ambito all'interno di un metodo.

Esercizio 11 (2) Create una classe interna **private** che implementa un'interfaccia **public**. Scrivete un metodo che restituisce un riferimento a un'istanza della classe interna **private**, sottoposta a upcast all'interfaccia. Dimostrate che la classe interna è completamente nascosta, cercando di eseguire un downcast a questa classe.

Classi interne anonime

L'esempio seguente sembrerà un po' insolito.

```
//: innerclasses/Parcel7.java
// Restituisce un'istanza di una classe interna anonima.

public class Parcel7 {
    public Contents contents() {
        return new Contents() { // Inserisce una definizione di
                               // classe
            private int i = 11;
            public int value() { return i; }
        }; // In questo caso il punto e virgola e' richiesto
    }
    public static void main(String[] args) {
        Parcel7 p = new Parcel7();
        Contents c = p.contents();
    }
} ///:~
```

Il metodo **contents()** combina la creazione del valore di ritorno con la definizione della classe che rappresenta quello stesso valore! Inoltre, la classe è anonima. Per complicare ancora un po' le cose, è come se iniziaste a creare un oggetto **Contents**, ma prima di arrivare al punto e virgola cambiaste idea, decidendo di introdurre una definizione di classe.

Ciò che questa strana sintassi vuole esprimere è "crea un oggetto di una classe anonima che abbia ereditato da **Contents**". Il riferimento restituito dalla nuova espressione viene sottoposto automaticamente a upcast a un



riferimento **Contents**. La sintassi della classe interna anonima è una forma breve di:

```
//: innerclasses/Parcel7b.java
// Versione espansa di Parcel7.java

public class Parcel7b {
    class MyContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    public Contents contents() { return new MyContents(); }
    public static void main(String[] args) {
        Parcel7b p = new Parcel7b();
        Contents c = p.contents();
    }
} //:~
```

Nella classe interna anonima **Contents** viene creato utilizzando un costruttore predefinito.

Il seguente codice mostra che cosa fare se la classe di base richiede un costruttore con un argomento.

```
//: innerclasses/Parcel8.java
// Chiamata al costruttore della classe di base.

public class Parcel8 {
    public Wrapping wrapping(int x) {
        // Chiamata al costruttore di base:
        return new Wrapping(x);
        // Passa un argomento al costruttore.
        public int value() {
            return super.value() * 47;
        }
    } // Il punto e virgola e' richiesto
}
```



```
public static void main(String[] args) {
    Parcel8 p = new Parcel8();
    Wrapping w = p.wrapping(10);
}
} //:~
```

In pratica si tratta di un semplice passaggio dell'argomento appropriato al costruttore della classe di base, in questo caso specifico espresso dalla **x** passata in **new Wrapping(x)**. Benché sia una classe comune con un'implementazione, **Wrapping** svolge anche il ruolo di “interfaccia” comune verso le sue classi derivate.

```
//: innerclasses/Wrapping.java
public class Wrapping {
    private int i;
    public Wrapping(int x) { i = x; }
    public int value() { return i; }
}
} //:~
```

Noterete che **Wrapping** ha un costruttore che richiede un argomento, per rendere le cose un po' più interessanti.

Il punto e virgola alla fine della classe interna anonima non indica la fine del corpo della classe bensì la fine dell'espressione che contiene la classe anonima: in questo senso, il suo utilizzo diventa quindi perfettamente standard.

Potete anche eseguire l'inizializzazione al momento di definire i campi in una classe anonima.

```
//: innerclasses/Parcel9.java
// Una classe interna anonima che esegue
// un'inizializzazione: e' una versione abbreviata di Parcel5.
// java.

public class Parcel9 {
```

```
    // Per essere utilizzato in una classe interna anonima
    // l'argomento deve essere final:
    public Destination destination(final String dest) {
        return new Destination() {
            private String label = dest;
```



```

    public String readLabel() { return label; }
};

public static void main(String[] args) {
    Parcel9 p = new Parcel9();
    Destination d = p.destination("Tasmania");
}
} //:~
```

Quando definite una classe interna anonima e desiderate servirvi di un oggetto esterno a essa, il riferimento all'argomento deve essere **final**, come l'argomento di **destination()** dell'esempio: se ve ne dimenticate il compilatore ve lo segnalerà con un messaggio di errore.

Finché si tratta di una semplice assegnazione di campo la tecnica adottata in questo esempio va benissimo, ma se voleste eseguire attività analoghe a quelle del costruttore?

Non è possibile avere un costruttore “nominativo” in una classe anonima (che è appunto priva di nome), tuttavia con l'inizializzazione di istanza potete, in effetti, generare un costruttore per una classe interna anonima, come nell'esempio seguente.

```

//: innerclasses/AnonymousConstructor.java
// Creazione di un costruttore per una classe interna anonima.
import static net.mindview.util.Print.*;

abstract class Base {
    public Base(int i) {
        print("Base constructor, i = " + i);
    }
    public abstract void f();
}

public class AnonymousConstructor {
    public static Base getBase(int i) {
        return new Base(i) {
            { print("Inside instance initializer"); }
        };
    }
}
```

```

    public void f() {
        print("In anonymous f()");
    }
}

public static void main(String[] args) {
    Base base = getBase(47);
    base.f();
}
} /* Output:
Base constructor, i = 47
Inside instance initializer
In anonymous f()
*///:~
```

In questo caso non si è dovuto indicare come **final** la variabile **i**: benché **i** sia passata al costruttore di base della classe anonima, non è mai utilizzata direttamente all'interno della classe anonima.

Di seguito è mostrato il tema di “parcel” con l'inizializzazione d'istanza; notate che gli argomenti di **destination()** devono essere **final**, poiché utilizzati nell'ambito della classe anonima.

```

//: innerclasses/Parcel10.java
// Utilizzo della "inizializzazione d'istanza" per eseguire
// la costruzione di una classe interna anonima.

public class Parcel10 {
    public Destination
    destination(final String dest, final float price) {
        return new Destination() {
            private int cost;
            // Inizializzazione d'istanza per ogni oggetto:
            {
                cost = Math.round(price);
                if(cost > 100)
                    System.out.println("Over budget!");
            }
        };
    }
}
```



```

private String label = dest;
public String readLabel() { return label; }
};

}

public static void main(String[] args) {
    Parcel10 p = new Parcel10();
    Destination d = p.destination("Tasmania", 101.395F);
}
/* Output:
Over budget!
*///:~

```

All'interno dell'inizializzatore d'istanza si trova il codice che non ha potuto essere eseguito come parte di un inizializzatore di campo, vale a dire l'istruzione **if**. Così, in effetti, gli inizializzatori d'istanza risultano essere i costruttori di classi interne anonime; naturalmente la loro portata è limitata: non essendo possibile sovraccaricarli, si può avere soltanto uno di questi costruttori.

Anche le classi interne anonime hanno funzionalità piuttosto ridotte, se paragonate alla normale ereditarietà, poiché possono estendere (con la parola chiave **extends**) una classe o implementare un'interfaccia, ma non eseguire entrambe le operazioni; inoltre può essere implementata una sola interfaccia.

Esercizio 12 (1) Ripetete l'Esercizio 7 utilizzando una classe interna anonima.

Esercizio 13 (1) Ripetete l'Esercizio 9 utilizzando una classe interna anonima.

Esercizio 14 (1) Modificate l'esempio **interfaces/HorrorShow.java** per implementare **DangerousMonster** e **Vampire** utilizzando classi anonime.

Esercizio 15 (2) Create una classe avente un costruttore non predefinito (che accetta argomenti); tale classe non dovrà contenere il costruttore predefinito (che non prevede argomenti). Create una seconda classe dotata di un metodo che restituisce un riferimento a un oggetto della prima classe. Costruite l'oggetto restituito creando una classe interna anonima che eredita dalla prima classe.



Nuove considerazioni su Factory Method

Osservate come l'esempio **interfaces/Factories.java** sia più elegante grazie alle classi interne anonime.

```

//: innerclasses/Factories.java
import static net.mindview.util.Print.*;

interface Service {
    void method1();
    void method2();
}

interface ServiceFactory {
    Service getService();
}

class Implementation1 implements Service {
    private Implementation1() {}
    public void method1() {print("Implementation1 method1");}
    public void method2() {print("Implementation1 method2");}
    public static ServiceFactory factory =
        new ServiceFactory() {
            public Service getService() {
                return new Implementation1();
            }
        };
}

class Implementation2 implements Service {
    private Implementation2() {}
    public void method1() {print("Implementation2 method1");}
    public void method2() {print("Implementation2 method2");}
    public static ServiceFactory factory =
        new ServiceFactory() {
            public Service getService() {

```



```

        return new Implementation2();
    }
}

public class Factories {
    public static void serviceConsumer(ServiceFactory fact) {
        Service s = fact.getService();
        s.method1();
        s.method2();
    }
    public static void main(String[] args) {
        serviceConsumer(Implementation1.factory);
        // Le implementazioni sono completamente intercambiabili:
        serviceConsumer(Implementation2.factory);
    }
} /* Output:
Implementation1 method1
Implementation1 method2
Implementation2 method1
Implementation2 metodo2
*//*:~
```

Ora i costruttori di **Implementation1** e **Implementation2** possono essere **private**, e non è più necessario creare una classe nominata di tipo factory.

Spesso si ha bisogno di un solo oggetto factory, che in questo esempio è stato creato come campo **static** nell'implementazione **Service**, con una sintassi di gran lunga più significativa.

Anche l'esempio **interfaces/Games.java** può essere migliorato per mezzo delle classi interne anonime.

```

//: innerclasses/Games.java
// Utilizzo delle classi interne anonime con il framework Game.
import static net.mindview.util.Print.*;

interface Game { boolean move(); }
interface GameFactory { Game getGame(); }
```

```

class Checkers implements Game {
    private Checkers() {}
    private int moves = 0;
    private static final int MOVES = 3;
    public boolean move() {
        print("Checkers move " + moves);
        return ++moves != MOVES;
    }
    public static GameFactory factory = new GameFactory() {
        public Game getGame() { return new Checkers(); }
    };
}

class Chess implements Game {
    private Chess() {}
    private int moves = 0;
    private static final int MOVES = 4;
    public boolean move() {
        print("Chess move " + moves);
        return ++moves != MOVES;
    }
    public static GameFactory factory = new GameFactory() {
        public Game getGame() { return new Chess(); }
    };
}

public class Games {
    public static void playGame(GameFactory factory) {
        Game s = factory.getGame();
        while(s.move())
            ;
    }
    public static void main(String[] args) {
        playGame(Checkers.factory);
        playGame(Chess.factory);
    }
} /* Output:
```

```
Checkers move 0
Checkers move 1
Checkers move 2
Chess move 0
Chess move 1
Chess move 2
Chess move 3
*/~/~
```

Fate tesoro del consiglio finale del capitolo precedente: preferite le classi alle interfacce. Se il vostro progetto richiede la presenza di un'interfaccia, ve ne renderete conto. In ogni caso evitate di servirvi delle interfacce fino a quando non siate costretti a farlo.

Esercizio 16 (1) Modificate la soluzione dell'Esercizio 18 del Capitolo 9 per utilizzare classi interne anonime.

Esercizio 17 (1) Modificate la soluzione dell'Esercizio 19 del Capitolo 9 per utilizzare classi interne anonime.

Classi nidificate

Se non avete bisogno di un collegamento tra l'oggetto di classe interna e quello di classe esterna potete rendere **static** la classe interna, creando in tal modo una cosiddetta *classe nidificata* (*nested class*).²

Per comprendere il significato della parola chiave **static** applicata alle classi interne, dovete ricordare che l'oggetto di una comune classe interna mantiene implicitamente un riferimento all'oggetto della classe esterna che l'ha creato. Questo non è vero, tuttavia, se rendete **static** una classe interna. Una classe nidificata implica che:

1. non avete bisogno di un oggetto di classe esterna per creare un oggetto di una classe nidificata;
2. non è possibile accedere a un oggetto di classe esterna non **static** da un oggetto di una classe nidificata.

Le classi nidificate sono diverse dalle comuni classi interne anche per un altro aspetto. I campi e i metodi delle normali classi interne possono essere solo al livello esterno di una classe, per cui le classi interne comuni non possono

2. La *nested class* è simile alla classe nidificata in C++, che tuttavia, a differenza di quelle Java, non può accedere a membri privati.

avere dati **static**, campi **static**, né classi nidificate. Tuttavia le classi nidificate possono contenere tutti questi elementi **static**.

```
//: innerclasses/Parcel11.java
// Classi nidificate (classi interne static).

public class Parcel11 {
    private static class ParcelContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    protected static class ParcelDestination
        implements Destination {
        private String label;
        private ParcelDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() { return label; }
        // Le classi nidificate possono
        // contenere altri elementi static:
        public static void f() {}
        static int x = 10;
        static class AnotherLevel {
            public static void f() {}
            static int x = 10;
        }
    }
    public static Destination destination(String s) {
        return new ParcelDestination(s);
    }
    public static Contents contents() {
        return new ParcelContents();
    }
    public static void main(String[] args) {
        Contents c = contents();
        Destination d = destination("Tasmania");
    }
} //~/~
```

In **main()** non è necessario alcun oggetto di **Parcel11**; viene invece utilizzata la normale sintassi per selezionare un membro **static** e chiamare i metodi che restituiscono riferimenti a **Contents** e **Destination**.

Come avete visto in precedenza, in una normale classe interna (non **static**) il collegamento all'oggetto di classe esterna si ottiene con uno speciale riferimento **this**. Una classe nidificata non possiede questo riferimento **this** e ciò la rende analoga a un metodo **static**.

Esercizio 18 (1) Create una classe contenente una classe nidificata e, in **main()**, generate un'istanza della classe nidificata.

Esercizio 19 (2) Create una classe che contenga una classe interna, a sua volta contenente un'altra classe interna. Ripetete l'esercitazione utilizzando classi nidificate: prendete nota dei nomi dei file **.class** generati dal compilatore.

Classi all'interno di interfacce

Normalmente non è permesso inserire codice in un'interfaccia, tuttavia una classe nidificata può fare parte di un'interfaccia. Qualsiasi classe inseriate in un'interfaccia è automaticamente **public** e **static**; essendo **static** non viola le regole delle interfacce: la classe nidificata è soltanto posta nell'ambito dello spazio di nomi dell'interfaccia. Potete anche implementare l'interfaccia circostante nella classe interna, come mostrato di seguito.

```
//: innerclasses/ClassInInterface.java
// {main: ClassInInterface$Test}

public interface ClassInInterface {
    void howdy();
}

class Test implements ClassInInterface {
    public void howdy() {
        System.out.println("Howdy!");
    }
}

public static void main(String[] args) {
    new Test().howdy();
}

} /* Output:
Howdy!
*///:~
```

Per creare codice comune da utilizzare con le varie implementazioni di un'interfaccia troverete particolarmente pratico nidificare una classe nell'interfaccia.

Nel corso del manuale è stato suggerito di inserire un metodo **main()** in ogni classe allo scopo di agevolare la fase di test. Un inconveniente di questa tattica è la maggiore quantità di codice compilato che verrà creato. Se ritenete che questo sia un problema potrete utilizzare una classe nidificata che contenga il codice necessario per i test.

```
//: innerclasses/TestBed.java
// Inserimento del codice in una classe nidificata.
// {main: TestBed$Tester}

public class TestBed {
    public void f() { System.out.println("f()"); }
    public static class Tester {
        public static void main(String[] args) {
            TestBed t = new TestBed();
            t.f();
        }
    }
} /* Output:
f()
*///:~
```

Questo esempio genera una classe separata chiamata **TestBed\$Tester**: per eseguire il programma digiterete il comando **java TestBed\$Tester**, tenendo conto che a seconda del sistema operativo utilizzato potreste dovere ricorrere alla tecnica di escape (**java TestBed\\$Tester** in Mac OS X e Unix/Linux).

Utilizzate questa classe per eseguire i test ma senza includerla nell'applicazione finale: cancellate **TestBed\$Tester.class** prima di approntare il pacchetto da distribuire.

Esercizio 20 (1) Create un'interfaccia che contiene una classe nidificata; implementate questa interfaccia e create un'istanza della classe nidificata.

Esercizio 21 (2) Create un'interfaccia che contiene una classe nidificata: essa deve possedere un metodo **static** che chiama i metodi della vostra interfaccia e ne visualizza i risultati. Implementate l'interfaccia e passate un'istanza della vostra implementazione al metodo.



Accesso dall'interno di classi a nidificazione multipla

Non importa quanto profondamente una classe interna possa essere nidificata; essa può accedere in modo trasparente a tutti i membri di tutte le classi all'interno delle quali è nidificata, come si evince dal seguente codice.³

```
//: innerclasses/MultiNestingAccess.java
// Le classi nidificate possono accedere a tutti i membri
// di tutti i livelli di classi in cui sono nidificate
class MNA {
    private void f() {}
    class A {
        private void g() {}
        public class B {
            void h() {
                g();
                f();
            }
        }
    }
}

public class MultiNestingAccess {
    public static void main(String[] args) {
        MNA mna = new MNA();
        MNA.A mnaa = mna.new A();
        MNA.A.B mnaab = mnaa.new B();
        mnaab.h();
    }
} //:~
```

Come potete vedere in **MNA.A.B**, i metodi **g()** e **f()** sono richiamabili senza alcun requisito, malgrado siano **private**. Questo esempio evidenzia anche la sintassi necessaria per creare oggetti di classi con livelli di nidificazione multipla, quando si creano oggetti in una classe diversa. La sintassi **.new** genera l'ambito corretto, per non dovere qualificare il nome della classe nella chiamata al costruttore.

³ L'autore ringrazia nuovamente Martin Danner per avere proposto l'argomento.

Perché utilizzare le classi interne?

A questo punto avete visto sintassi e semantica che descrivono il funzionamento delle classi interne, ma ciò non spiega i motivi della loro esistenza. Perché i progettisti di Java si sono disturbati ad aggiungere questa caratteristica al linguaggio?

Di solito la classe interna eredita da una classe o implementa un'interfaccia e il codice presente nella classe interna manipola l'oggetto nella classe esterna all'interno della quale è stato creato. Si potrebbe affermare che una classe interna fornisce una specie di "finestra" sulla classe esterna.

Una domanda che potreste porvi è la seguente: "se occorre soltanto un riferimento a un'interfaccia, perché non lasciare che venga implementata dalla classe esterna?"; la risposta è che, se questo è tutto ciò di cui avete bisogno, allora è il metodo da utilizzare. Dunque, che cosa distingue una classe interna che implementa un'interfaccia, da una classe esterna che implementa la stessa interfaccia? La risposta è che non sempre è possibile utilizzare le interfacce, e talvolta occorre lavorare con le implementazioni. Quindi il motivo essenziale che giustifica l'utilizzo delle classi interne è il seguente:

Ogni classe interna può ereditare indipendentemente da un'implementazione. Di conseguenza la classe interna non è limitata dal fatto che la classe esterna stia già ereditando da un'implementazione.

Senza la caratteristica che consente alle classi interne di ereditare da più classi reali o **abstract**, alcuni problemi di programmazione e progettazione sarebbero insormontabili. Pertanto, uno dei punti di vista da cui si può considerare la classe interna è come complemento alla soluzione del problema dell'ereditarietà multipla. Le interfacce risolvono una parte del problema, mentre le classi interne permettono di ottenere efficacemente un'"ereditarietà di implementazione multipla". In pratica, le classi interne consentono effettivamente di ereditare da più entità "non-interfaccia".

Per chiarire questo concetto, immaginate di disporre di due interfacce che dovete in qualche modo implementare all'interno di una classe. Grazie alla flessibilità delle interfacce sono possibili due scelte: la creazione di una singola classe o di una classe interna.

```
//: innerclasses/MultiInterfaces.java
// Due metodi mediante i quali una classe puo' implementare
// interfacce multiple
package innerclasses;
```



```

interface A {}

interface B {}

class X implements A, B {}

class Y implements A {
    B makeB() {
        // Classe interna anonima:
        return new B() {};
    }
}

public class MultiInterfaces {
    static void takesA(A a) {}
    static void takesB(B b) {}
    public static void main(String[] args) {
        X x = new X();
        Y y = new Y();
        takesA(x);
        takesA(y);
        takesB(x);
        takesB(y.makeB());
    }
} //:~

```

Ovviamente questo presuppone che la struttura del vostro codice abbia senso in un caso o nell'altro. Di norma, per scegliere se utilizzare una singola classe o una classe interna avrete qualche indicazione sulla natura del problema da risolvere; in assenza di vincoli, tuttavia, l'approccio illustrato nell'esempio precedente non fa differenza dal punto di vista dell'implementazione: entrambi funzionano.

Di contro, se utilizzate classi reali o **abstract** invece delle interfacce, sarete vincolati all'utilizzo di classi interne qualora la vostra classe debba implementare entrambe le altre.

```

//: innerclasses/MultiImplementation.java
// Con classi reali o astratte, le classi interne sono il solo
// mezzo

```



```

// per ottenere l'effetto di "ereditarietà di implementazione
// multipla"
package innerclasses;

class D {}

abstract class E {}

class Z extends D {
    E makeE() { return new E(); }
}

public class MultiImplementation {
    static void takesD(D d) {}
    static void takesE(E e) {}
    public static void main(String[] args) {
        Z z = new Z();
        takesD(z);
        takesE(z.makeE());
    }
} //:~

```

Se non avete bisogno di risolvere il problema dell'“ereditarietà di implementazione multipla” probabilmente potrete gestire il vostro codice senza bisogno di utilizzare classi interne. Le classi interne, però, offrono alcune caratteristiche aggiuntive.

1. La classe interna può avere istanze multiple, ognuna con le proprie informazioni di stato, indipendenti da quelle dell'oggetto di classe esterna.
2. Una classe esterna può avere numerose classi interne, ognuna delle quali implementa la stessa interfaccia o eredita dalla stessa classe in modo diverso. Vedrete tra breve un esempio di questo meccanismo.
3. Il punto di creazione dell'oggetto di classe interna non è legato alla creazione dell'oggetto di classe esterna.
4. Non si corre il rischio di confondere la relazione “è-un” con la classe interna, poiché si tratta di entità separate.

Per esempio, se **Sequence.java** non utilizzasse classi interne, dovreste affermare che “una **Sequence** è un **Selector**”, e per una determinata **Se-**

quence potrebbe esistere un solo **Selector**; ma potreste anche avere un secondo metodo, **reverseSelector()**, che produce un **Selector** che si sposta attraverso la sequenza. Questo tipo di flessibilità è offerto soltanto dalle classi interne.

Esercizio 22 (2) Implementate **reverseSelector()** in **Sequence.java**.

Esercizio 23 (4) Create un'interfaccia **U** contenente tre metodi. Poi create una classe **A** con un metodo che genera un riferimento a **U** tramite una classe interna anonima, quindi una seconda classe **B** che contiene un array di **U**: **B** dovrebbe avere un metodo che accetta e memorizza nell'array un riferimento a **U**, un secondo metodo che imposta a **null** un riferimento (specificato dall'argomento del metodo) contenuto nell'array, e un terzo metodo che si sposta attraverso l'array e chiama i metodi in **U**. In **main()** create un gruppo di oggetti **A** e un oggetto **B**: popolate **B** con i riferimenti a **U** prodotti dagli oggetti **A** e utilizzate **B** per richiamare tutti gli oggetti **A**. Rimuovete alcuni riferimenti di **U** da **B**.

Closure e callback

Si definisce *closure*, letteralmente “chiusura”, un oggetto richiamabile che mantiene le informazioni dell’ambito in cui è stato creato. Da questa definizione appare evidente che una classe interna è una closure orientata agli oggetti, poiché non si limita a contenere le informazioni dell’oggetto di classe esterna (“l’ambito in cui è stato creato”), ma mantiene automaticamente un riferimento all’intero oggetto di classe esterna, nei confronti del quale è autorizzata a manipolare tutti i membri, anche se **private**.

Uno dei motivi che hanno giustificato l’esigenza di gestire puntatori in Java è stato quello di permettere le *callback*.

Il meccanismo di callback è molto semplice: a un oggetto vengono fornite informazioni che gli permettono di richiamare l’oggetto di origine in qualsiasi momento.

Come vedrete nel corso della trattazione, si tratta di un concetto molto potente. Se una callback viene implementata utilizzando un puntatore, tuttavia, dovete fare affidamento sul fatto che il programmatore operi in modo corretto e non usi il puntatore impropriamente.

Vi sarete ormai resi conto che Java adotta un approccio molto cautelativo, al punto che i puntatori non sono stati previsti nel linguaggio.

La funzionalità di closure fornita dalla classe interna è un’eccellente soluzione: più flessibile e certamente più sicura di un puntatore. Considerate l’esempio seguente.

```
//: innerclasses/Callbacks.java
// Utilizzo delle classi interne per le callback
package innerclasses;
import static net.mindview.util.Print.*;

interface Incrementable {
    void increment();
}

// Molto semplice, soltanto per implementare l'interfaccia:
class Callee1 implements Incrementable {
    private int i = 0;
    public void increment() {
        i++;
        print(i);
    }
}

class MyIncrement {
    public void increment() { print("Other operation"); }
    static void f(MyIncrement mi) { mi.increment(); }
}

// Se la classe deve implementare increment() in qualche
// altro modo, dovete ricorrere a una classe interna:
class Callee2 extends MyIncrement {
    private int i = 0;
    public void increment() {
        super.increment();
        i++;
        print(i);
    }
}

private class Closure implements Incrementable {
```

```

public void increment() {
    // Specifica il metodo della classe esterna, altrimenti
    // si otterebbe un'iterazione infinita:
    Callee2.this.increment();
}

Incrementable getCallbackReference() {
    return new Closure();
}

class Caller {
    private Incrementable callbackReference;
    Caller(Incrementable cbh) { callbackReference = cbh; }
    void go() { callbackReference.increment(); }
}

public class Callbacks {
    public static void main(String[] args) {
        Callee1 c1 = new Callee1();
        Callee2 c2 = new Callee2();
        MyIncrement.f(c2);
        Caller caller1 = new Caller(c1);
        Caller caller2 = new Caller(c2.getCallbackReference());
        caller1.go();
        caller1.go();
        caller2.go();
        caller2.go();
    }
} /* Output:
Other operation
1
1
2
Other operation
2
)

```

Other operation

3
*///:~

Questo codice mostra anche un'altra distinzione tra l'implementazione di un'interfaccia in una classe esterna rispetto a quella in una classe interna. **Callee1** è chiaramente la soluzione più semplice in termini di codice. **Callee2** eredita da **MyIncrement**, il quale ha già un diverso metodo **increment()** che fa qualcosa di diverso rispetto a quanto si attende l'interfaccia **Incrementable**. Quando **MyIncrement** viene ereditato in **Callee2**, **increment()** non può essere sovrascritto per essere utilizzato da **Incrementable**, pertanto siete costretti a fornire un'implementazione separata mediante una classe interna. Notate anche che quando create una classe interna non aggiungete né modificate l'interfaccia della classe esterna.

In **Callee2** tutto è **private** tranne il metodo **getCallbackReference()**. Per consentire qualsiasi collegamento al mondo esterno, l'interfaccia **Incrementable** è essenziale. In questo codice potete vedere che le interfacce permettono una separazione netta tra interfaccia e implementazione.

La classe interna **Closure** implementa **Incrementable** per fornire un riferimento sicuro a **Callee2**. Naturalmente, qualsiasi oggetto ottenga il riferimento **Incrementable** può chiamare solo **increment()** e non ha alcuna altra capacità, a differenza di un puntatore che permetterebbe di eseguire qualsiasi operazione.

Il costruttore di **Caller** accetta un riferimento **Incrementable**, sebbene l'intercettazione del riferimento di callback possa avvenire in ogni momento: quindi, poco dopo, utilizza il riferimento per "richiamare" (*to call back*, appunto) la classe **Callee**.

Il valore della callback è la sua flessibilità poiché permette di decidere in modo dinamico quali metodi verranno chiamati in fase di esecuzione. Il vantaggio di questa caratteristica diventerà più evidente nel Volume 3, Capitolo 2, dove le callback sono utilizzate diffusamente per implementare le funzionalità dell'interfaccia grafica.

Classi interne e framework di controllo

Un esempio più concreto dell'impiego delle classi interne si riscontra in una struttura a cui si farà riferimento con l'espressione "framework di controllo".

Un framework applicativo (*application framework*) è una classe o insieme di classi progettati per risolvere un tipo di problema specifico. Di solito, per

impostare un framework applicativo ereditate da una o più classi e sovrascrivete alcuni metodi messi a disposizione dal framework. Il codice che scrivete nei metodi sovrascritti “personalizza” la soluzione generica fornita dal framework applicativo, al fine di risolvere il problema specifico: un esempio del design pattern *Template Method*, di cui troverete maggiori informazioni su *Thinking in Patterns (with Java)*, disponibile all’indirizzo www.mindview.net. Il Template Method contiene la struttura di base dell’algoritmo e chiama uno o più metodi sovrascrivibili, che completano l’azione dell’algoritmo stesso. Un design pattern separa i componenti variabili da quelli costanti; in questo caso Template Method è l’elemento costante, mentre i metodi sovrascrivibili rappresentano la parte che viene modificata.

Un framework di controllo è un tipo di struttura applicativa particolare caratterizzata dalla necessità di rispondere agli eventi, in ciò che si definisce sistema *event-driven*, letteralmente “guidato dagli eventi”. Un tipico problema nella programmazione applicativa è rappresentato dall’interfaccia grafica utente (GUI, *Graphic User Interface*), che è quasi interamente event-driven. Come vedrete nel Volume 3, Capitolo 2, la libreria Java Swing è un framework di controllo che risolve elegantemente il problema delle GUI, ricorrendo in modo estensivo alle classi interne.

Per analizzare il funzionamento di queste strutture considerate un framework di controllo incaricato di gestire gli eventi che si trovano nello stato di “ready”: benché la condizione di “ready” possa significare qualsiasi cosa, ai fini di questo esempio la si considera basata sul clock di sistema.

Di seguito è simulato un framework di controllo che non contiene alcuna informazione specifica sugli eventi che controlla: queste informazioni sono fornite dall’ereditarietà, quando viene implementata la porzione **action()** dell’algoritmo.

Per prima cosa considerate l’interfaccia che descrive qualsiasi evento di controllo. Si tratta di una classe **abstract**, non di un’interfaccia effettiva, poiché il comportamento predefinito comporta l’esecuzione di un controllo basato sul tempo. Una parte dell’implementazione è mostrata di seguito.

```
//: innerclasses/controller/Event.java
// I metodi comuni a tutti gli eventi di controllo.
package innerclasses.controller;

public abstract class Event {
    private long eventTime;
    protected final long delayTime;
```

```
public Event(long delayTime) {
    this.delayTime = delayTime;
    start();
}

public void start() { // Permette il riavvio
    eventTime = System.nanoTime() + delayTime;
}

public boolean ready() {
    return System.nanoTime() >= eventTime;
}

public abstract void action();
} ///:~
```

Il costruttore carica il “ritardo” (*delayTime*) misurato a partire dalla creazione dell’oggetto, trascorso il quale **Event** dovrà essere eseguito, poi chiama **start()** che intercetta il momento corrente (*System.nanoTime()*) e vi aggiunge il ritardo necessario per calcolare il momento in cui si verificherà l’evento (*eventTime*). Anziché essere incluso nel costruttore, il metodo **start()** è mantenuto separato: questo consente di riavviare il timer dopo che l’evento è stato eseguito, in modo che l’oggetto **Event** possa essere riutilizzato. Per esempio, se voleste produrre un evento ricorrente, basterebbe chiamare **start()** nel vostro metodo **action()**.

Il metodo **ready()** informa quando è il momento di eseguire il metodo **action()**; naturalmente **ready()** può essere sovrascritto in una classe derivata per consentirvi di basare **Event** su criteri diversi dal tempo.

Il seguente file contiene il framework di controllo effettivo che gestisce e attiva gli eventi. Gli oggetti **Event** sono contenuti in un oggetto contenitore di tipo **List<Event>** (pronunciato come “List of Events”, elenco di eventi), che vedrete in dettaglio nel Capitolo 11: per il momento vi basterà sapere che **add()** accoda un **Event** all’oggetto **List**; **size()** restituisce il numero di voci presenti nell’oggetto **List**; la sintassi *foreach* itera consecutivamente gli oggetti **Event** da **List** e il metodo **remove()** elimina da **List** l’**Event** specificato.

```
//: innerclasses/controller/Controller.java
// Framework riutilizzabile per i sistemi di controllo.
package innerclasses.controller;

import java.util.*;
```



```

public class Controller {
    // Una classe di java.util per contenere oggetti Event:
    private List<Event> eventList = new ArrayList<Event>();
    public void addEvent(Event c) { eventList.add(c); }
    public void run() {
        while(eventList.size() > 0)
            // Esegue una copia, in modo da non modificare l'elenco
            // quando ne vengono selezionati gli elementi:
            for(Event e : new ArrayList<Event>(eventList))
                if(e.ready()) {
                    System.out.println(e);
                    e.action();
                    eventList.remove(e);
                }
    }
} //:~

```

Il metodo **run()** itera una copia di **eventList** cercando un oggetto **Event** che sia **ready()**, vale a dire pronto per essere eseguito; per ogni **Event** che abbia la condizione **ready()** visualizza informazioni, utilizzando il metodo **toString()** dell'oggetto, chiama il metodo **action()** e rimuove infine l'**Event** dall'elenco **List**.

È interessante notare che finora in questo progetto non è assolutamente chiaro quale sia il compito svolto da **Event**. Questa è appunto la chiave della progettazione: “separare ciò che cambia da quanto rimane costante”. Per usare la definizione dell'autore, il “vettore di modifica” è rappresentato dalle diverse azioni svolte dai vari tipi di oggetti **Event**; esprimete azioni differenti creando diverse sottoclassi di **Event**.

È a questo punto che entrano in gioco le classi interne, che consentono di:

1. creare in una sola classe l'intera implementazione di un framework di controllo, incapsulando tutti i componenti che sono univoci nell'applicazione corrente; le classi interne vengono utilizzate per esprimere i vari tipi di **action()** necessari per risolvere il problema;
2. evitare che l'implementazione diventi oltremodo complessa, dal momento che permettono di accedere facilmente a qualsiasi membro della classe esterna; senza questa caratteristica il codice potrebbe diventare difficile da coordinare, al punto tale che sareste costretti a cercare un'alternativa.

Considerate una particolare implementazione del framework di controllo progettata per controllare le funzioni di una serra.



Ogni azione è diversa: luci, apertura di acqua e attivazione dei termostati, accensione dei segnali acustici e riavvio dell'applicazione. Il framework di controllo, tuttavia, è progettato per isolare facilmente questo codice così diverso.

Grazie alle classi interne potete disporre di più versioni derivate dalla stessa classe di base, **Event**, nell'ambito di una sola classe. Per ogni tipo di azione ereditate una nuova classe interna **Event** e scrivete il codice di controllo nell'implementazione di **action()**.

Come avviene comunemente in un framework applicativo, la classe **GreenhouseControls** viene ereditata da **Controller**.

```

//: innerclasses/GreenhouseControls.java
// Questo esempio crea un'applicazione specifica del sistema
// di controllo,
// contenuto in una sola classe. Le classi interne permettono
// di incapsulare diverse funzionalita' per ogni tipo di
// evento.

```

```

import innerclasses.controller.*;

public class GreenhouseControls extends Controller {
    private boolean light = false;
    public class LightOn extends Event {
        public LightOn(long delayTime) { super(delayTime); }
        public void action() {
            // Inserire qui il codice di controllo hardware
            // per accendere le luci.
            light = true;
        }
        public String toString() { return "Light is on"; }
    }
    public class LightOff extends Event {
        public LightOff(long delayTime) { super(delayTime); }
        public void action() {
            // Inserire qui il codice di controllo hardware
            // per spegnere le luci.
            light = false;
        }
    }
}

```



```
}

    public String toString() { return "Light is off"; }

}

private boolean water = false;
public class WaterOn extends Event {
    public WaterOn(long delayTime) { super(delayTime); }
    public void action() {
        // Inserire qui il codice di controllo hardware.
        water = true;
    }
    public String toString() {
        return "Greenhouse water is on";
    }
}

public class WaterOff extends Event {
    public WaterOff(long delayTime) { super(delayTime); }
    public void action() {
        // Inserire qui il codice di controllo hardware.
        water = false;
    }
    public String toString() {
        return "Greenhouse water is off";
    }
}

private String thermostat = "Day";
public class ThermostatNight extends Event {
    public ThermostatNight(long delayTime) {
        super(delayTime);
    }
    public void action() {
        // Inserire qui il codice di controllo hardware.
        thermostat = "Night";
    }
    public String toString() {
        return "Thermostat on night setting";
    }
}
```



```
}

public class ThermostatDay extends Event {
    public ThermostatDay(long delayTime) {
        super(delayTime);
    }
    public void action() {
        // Inserire qui il codice di controllo hardware.
        thermostat = "Day";
    }
    public String toString() {
        return "Thermostat on day setting";
    }
}

// Un esempio di action() che inserisce una ricorrenza
// di se stessa nell'elenco di eventi:
public class Bell extends Event {
    public Bell(long delayTime) { super(delayTime); }
    public void action() {
        addEvent(new Bell(delayTime));
    }
    public String toString() { return "Bing!"; }
}

public class Restart extends Event {
    private Event[] eventList;
    public Restart(long delayTime, Event[] eventList) {
        super(delayTime);
        this.eventList = eventList;
        for(Event e : eventList)
            addEvent(e);
    }
    public void action() {
        for(Event e : eventList) {
            e.start(); // Esegue di nuovo ogni evento
            addEvent(e);
        }
        start(); // Esegue di nuovo l'evento corrente
    }
}
```



```

        addEvent(this);
    }

    public String toString() {
        return "Restarting system";
    }

}

public static class Terminate extends Event {
    public Terminate(long delayTime) { super(delayTime); }
    public void action() { System.exit(0); }
    public String toString() { return "Terminating"; }
}

} //:~

```

Noteate che malgrado **light**, **water** e **thermostat** appartengano alla classe esterna **GreenhouseControls**, le classi interne possono accedere ai suoi campi senza requisiti o autorizzazioni speciali; inoltre i metodi **action()** normalmente comportano qualche genere di controllo hardware.

Quasi tutte le classi **Event** sono simili, tuttavia **Bell** e **Restart** sono speciali. **Bell** emette un suono e aggiunge un nuovo oggetto **Bell** all'elenco degli eventi, in modo da poter suonare nuovamente in seguito. Osservate come le classi interne siano simili all'ereditarietà multipla: **Bell** e **Restart** hanno tutti i metodi di **Event** e si comportano come se avessero anche tutti quelli della classe esterna **GreenhouseControls**.

A **Restart** viene passato un array di oggetti **Event** che aggiunge al controller; poiché **Restart()** è soltanto un altro oggetto **Event** potete anche aggiungere un oggetto **Restart** all'interno di **Restart.action()**, affinché l'applicazione si riavvii regolarmente.

La seguente classe configura il sistema creando un oggetto **GreenhouseControls** e aggiunge diversi tipi di oggetti **Event**. Questo è un esempio del design pattern *Command*, nel quale ogni oggetto in **eventList** è una richiesta incapsulata come oggetto.

```

//: innerclasses/GreenhouseController.java
// Configura ed esegue l'applicazione Greenhouse.
// {Args: 5000}
public class GreenhouseController {
    public static void main(String[] args) {
        GreenhouseControls gc = new GreenhouseControls();
        // Invece di codificare gli eventi, qui potete
        // caricarli da un file di configurazione
        gc.addEvent(gc.new Bell(900));
        Event[] eventList = {
            gc.new ThermostatNight(0),
            gc.new LightOn(200),
            gc.new LightOff(400),
            gc.new WaterOn(600),
            gc.new WaterOff(800),
            gc.new ThermostatDay(1400)
        };
        gc.addEvent(gc.new Restart(2000, eventList));
        if(args.length == 1)
            gc.addEvent(
                new GreenhouseControls.Terminate(
                    new Integer(args[0])));
        gc.run();
    }
} /* Output:
Bing!
Thermostat on night setting
Light is on
Light is off
Greenhouse water is on
Greenhouse water is off
Thermostat on day setting
Restarting system
Terminating
*///:~

```

```

public class GreenhouseController {
    public static void main(String[] args) {
        GreenhouseControls gc = new GreenhouseControls();
        // Invece di codificare gli eventi, qui potete
        // caricarli da un file di configurazione
        gc.addEvent(gc.new Bell(900));
        Event[] eventList = {
            gc.new ThermostatNight(0),
            gc.new LightOn(200),
            gc.new LightOff(400),
            gc.new WaterOn(600),
            gc.new WaterOff(800),
            gc.new ThermostatDay(1400)
        };
        gc.addEvent(gc.new Restart(2000, eventList));
        if(args.length == 1)
            gc.addEvent(
                new GreenhouseControls.Terminate(
                    new Integer(args[0])));
        gc.run();
    }
} /* Output:
Bing!
Thermostat on night setting
Light is on
Light is off
Greenhouse water is on
Greenhouse water is off
Thermostat on day setting
Restarting system
Terminating
*///:~

```

Questa classe inizializza il sistema, pertanto aggiunge tutti gli eventi appropriati. L'evento **Restart** viene eseguito ripetutamente e carica la **eventList** nell'oggetto **GreenhouseControls** a ogni ciclo. Da riga di comando potete



fornire un argomento che indica il numero di millisecondi trascorsi i quali il programma dovrà terminare: questa tecnica è utilizzata per i test.

Naturalmente, fare in modo che il programma legga gli eventi da un file di configurazione è più flessibile che codificarli nel codice: un esercizio del Volume 2, Capitolo 6 prevede che eseguiate esattamente questa operazione.

Il listato precedente dovrebbe consentirvi di apprezzare maggiormente le classi interne, soprattutto se utilizzate in un framework di controllo. Nel Volume 3, Capitolo 2 vedrete un modo elegante per utilizzare le classi interne al fine di descrivere le azioni di un'interfaccia grafica, che riuscirà a convincervi del tutto.

Esercizio 24 (2) In **GreenhouseControls.java** aggiungete alcune classi interne **Event** che accendono e spengono ventilatori, e configurate **GreenhouseController.java** per utilizzare i nuovi oggetti **Event**.

Esercizio 25 (3) Ereditate dalla classe **GreenhouseControls** in **GreenhouseControls.java**, per aggiungere classi interne **Event** che accendono e spengono alcuni vaporizzatori. Scrivete una nuova versione di **GreenhouseController.java** che si serva dei nuovi oggetti **Event**.

Come ereditare dalle classi interne

Poiché il costruttore della classe interna deve essere connesso a un riferimento all'oggetto della classe esterna, tutto diventa leggermente più complesso quando si eredita da una classe interna. Il problema è che il riferimento “segreto” all'oggetto della classe esterna deve essere inizializzato, tuttavia nella classe derivata non è più presente un oggetto predefinito al quale collegarsi. Per rendere esplicita l'associazione dovete ricorrere a una sintassi apposita.

```
//: innerclasses/InheritInner.java
// Ereditare una classe interna.

class WithInner {
    class Inner {}
}

public class InheritInner extends WithInner.Inner {
    //! InheritInner() {} // Non compila.
    InheritInner(WithInner wi) {
        wi.super();
    }
}
```

```
} //:~
```

Potete vedere che **InheritInner** estende solo la classe interna, non quella esterna. Ma quando sarà il momento di creare un costruttore, quello predefinito risulterà inadatto e non potrete semplicemente passare un riferimento a un oggetto inclusivo (esterno); dovete anche utilizzare la sintassi

```
enclosingClassReference.super();
```

nel costruttore. Questa operazione fornirà il riferimento necessario e a quel punto il programma verrà compilato senza problemi.

Esercizio 26 (2) Create una classe contenente una classe interna che possiede un costruttore non predefinito, vale a dire un costruttore che accetta argomenti, poi create una seconda classe con una classe interna che eredita dalla prima classe interna.

È possibile sovrascrivere le classi interne?

Potreste chiedervi che cosa accade quando create una classe interna, poi ereditate dalla classe inclusiva e ridefinite la classe interna: in pratica se è possibile “sovrascrivere” l'intera classe interna. Questo sembra essere un concetto molto potente, tuttavia l'azione di “sovrascrivere” una classe interna come se fosse un metodo della classe esterna non produce alcunché.

```
//: innerclasses/BigEgg.java
// Una classe interna non puo' essere sovrascritta come un
// metodo.
import static net.mindview.util.Print.*;

class Egg {
    private Yolk y;
    protected class Yolk {
        public Yolk() { print("Egg.Yolk()"); }
    }
}
```

```

}

public Egg() {
    print("New Egg()");
    y = new Yolk();
}
}

public class BigEgg extends Egg {
    public class Yolk {
        public Yolk() { print("BigEgg.Yolk()"); }
    }
    public static void main(String[] args) {
        new BigEgg();
    }
} /* Output:
New Egg()
Egg.Yolk()
*//*:~
```

Il costruttore predefinito viene creato automaticamente dal compilatore e chiama il costruttore della classe di base. Potreste pensare che dal momento che viene creato un oggetto **BigEgg** avrebbe dovuto essere utilizzata la versione “sovrascritta” di **Yolk**, tuttavia l’output dimostra che non è così.

L’esempio evidenzia che nelle classi interne non esiste alcuna “magia” particolare che si manifesti quando ereditate dalla classe esterna. Le due classi interne sono entità separate, ciascuna nel proprio spazio di nomi; è comunque ancora possibile ereditare esplicitamente dalla classe interna.

```
//: innerclasses/BigEgg2.java
// Ereditarieta' corretta di una classe interna.
import static net.mindview.util.Print.*;
```

```

class Egg2 {
    protected class Yolk {
        public Yolk() { print("Egg2.Yolk()"); }
        public void f() { print("Egg2.Yolk.f()"); }
    }
}
```

```

private Yolk y = new Yolk();
public Egg2() { print("New Egg2()"); }
public void insertYolk(Yolk yy) { y = yy; }
public void g() { y.f(); }
}

public class BigEgg2 extends Egg2 {
    public class Yolk extends Egg2.Yolk {
        public Yolk() { print("BigEgg2.Yolk()"); }
        public void f() { print("BigEgg2.Yolk.f()"); }
    }
    public BigEgg2() { insertYolk(new Yolk()); }
    public static void main(String[] args) {
        Egg2 e2 = new BigEgg2();
        e2.g();
    }
} /* Output:
Egg2.Yolk()
New Egg2()
Egg2.Yolk()
BigEgg2.Yolk()
BigEgg2.Yolk.f()
*//*:~
```

Ora **BigEgg2.Yolk** estende esplicitamente **Egg2.Yolk** e sovrascrive i suoi metodi. Il metodo **insertYolk()** permette a **BigEgg2** di eseguire l’upcast di uno dei suoi oggetti **Yolk** al riferimento **y** in **Egg2**, in modo che quando il metodo **g()** chiama il metodo **y.f()** venga utilizzata la versione sovrascritta di **f()**. La seconda chiamata a **Egg2.Yolk()** è la chiamata al costruttore della classe di base del costruttore di **BigEgg2.Yolk**: notate che chiamando **g()** viene utilizzata la versione sovrascritta di **f()**.

Classi interne locali

Come sapete, le classi interne possono essere create in blocchi di codice, generalmente nel corpo di un metodo. Una classe interna locale non può avere un modificatore d’accesso poiché non fa parte della classe esterna, tuttavia accede alle variabili di tipo **final** nel blocco di codice corrente e a tutti i mem-

bri della classe inclusiva. L'esempio seguente confronta la creazione di una classe interna locale con quella di una anonima.

```
//: innerclasses/LocalInnerClass.java
// Contiene una sequenza di Objects.
import static net.mindview.util.Print.*;

interface Counter {
    int next();
}

public class LocalInnerClass {
    private int count = 0;
    Counter getCounter(final String name) {
        // Una classe interna locale:
        class LocalCounter implements Counter {
            public LocalCounter() {
                // La classe interna locale puo' avere un costruttore
                print("LocalCounter()");
            }
            public int next() {
                printnb(name); // Accesso a final locale
                return count++;
            }
        }
        return new LocalCounter();
    }
    // La stessa funzionalita', ottenuta mediante una classe
    // interna anonima:
    Counter getCounter2(final String name) {
        return new Counter() {
            // Una classe interna anonima non puo' avere un
            // costruttore nominato, ma soltanto un inizializzatore
            // d'istanza:
            {
                print("Counter()");
            }
        };
    }
}
```

```
}
public int next() {
    printnb(name); // Accesso a final locale
    return count++;
}
};

public static void main(String[] args) {
    LocalInnerClass lic = new LocalInnerClass();
    Counter
    c1 = lic.getCounter("Local inner ");
    c2 = lic.getCounter2("Anonymous inner ");
    for(int i = 0; i < 5; i++)
        print(c1.next());
    for(int i = 0; i < 5; i++)
        print(c2.next());
}
} /* Output:
LocalCounter()
Counter()
Locale interna 0
Local inner 1
Local inner 2
Local inner 3
Local inner 4
Anonymous inner 5
Anonymous inner 6
Anonymous inner 7
Anonymous inner 8
Anonymous inner 9
*///:~
```

Counter restituisce il valore successivo in una sequenza: questo oggetto è implementato sia come classe locale sia come classe interna anonima, entrambe con gli stessi comportamenti e possibilità. Poiché il nome della classe interna locale non è accessibile fuori dal metodo, una delle ragioni per ricorrere a una classe interna locale in luogo di una anonima è la disponibilità di un



costruttore nominato e/o sovraccaricato: ricordate che la classe interna anonima può utilizzare soltanto l'inizializzazione d'istanza.

Un altro motivo per preferire una classe interna locale a una anonima è la necessità di creare oggetti multipli di quella classe.

Identificatori delle classi interne

Ogni classe produce un file **.class** che contiene tutte le informazioni sulle modalità di creazione degli oggetti di quel tipo: queste informazioni creano una “meta-classe”, detta oggetto **Class**. Da quanto detto potete intuire che le classi interne producono anch'esse file **.class** per contenere le informazioni relative ai propri oggetti **Class**. La nomenclatura di questi file/classi è regolata rigidamente: il nome della classe inclusiva, seguito dal simbolo ‘\$’, seguito a sua volta dal nome della classe interna. Per esempio, i file **.class** creati da **LocalInnerClass.java** sono:

```
Counter.class  
LocalInnerClass$1.class  
LocalInnerClass$1LocalCounter.class  
LocalInnerClass.class
```

Se le classi interne sono anonime, il compilatore si limiterà a generare numeri che fungono da identificativi per le classi interne. Invece, se le classi interne sono annidate all'interno di altre classi interne, i loro nomi verranno aggiunti dopo un simbolo “\$” e l'identificativo o gli identificativi delle classi che le contengono.

Nonostante questo schema di generazione dei nomi sia così semplice e diretto è estremamente affidabile, e gestisce in modo corretto la maggior parte delle situazioni.⁴

Poiché questo è lo schema di denominazione predefinito di Java, i file generati sono automaticamente indipendenti dalla piattaforma. Ricordate che il compilatore Java modifica le vostre classi interne in vari modi per farle funzionare correttamente.

4. Tenete presente che “\$” è un metacarattere della shell Unix, che può dare problemi in fase di elencazione dei file **.class**.

Riepilogo

Le interfacce e le classi interne sono tra i concetti più elaborati che troverete nella maggior parte dei linguaggi OOP: per esempio, in C++ non esiste niente di analogo. Insieme esse risolvono lo stesso problema che C++ tenta di risolvere con la sua funzionalità di ereditarietà multipla (MI, *Multiple Inheritance*). Tuttavia l'ereditarietà multipla in C++ è piuttosto difficile da utilizzare, mentre le interfacce Java e le classi interne sono molto più accessibili.

Benché le caratteristiche in sé siano ragionevolmente chiare, la loro applicazione è un argomento da definire in sede di progettazione, quasi quanto il polimorfismo. Con la pratica migliorerete la vostra capacità di riconoscere le situazioni in cui dovreste ricorrere a un'interfaccia, una classe interna o a entrambe le soluzioni.

In ogni caso a questo punto del manuale dovreste avere compreso agevolmente la sintassi e la semantica associate: tenete presente che in Java vedrete spesso queste caratteristiche in azione, pertanto non farete fatica ad assimilarle.

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

Capitolo 11

Come contenere gli oggetti



Un programma molto semplice ha una quantità fissa di oggetti, di durata nota.

Di solito i vostri programmi creeranno nuovi oggetti basandosi su alcuni criteri che saranno noti unicamente in fase di esecuzione, e prima di quel momento non conoscerete né la quantità né i tipi esatti degli oggetti che vi occorreranno. Questo è un generico problema di programmazione, per risolvere il quale è necessario potere creare qualsiasi numero di oggetti, in qualsiasi momento e punto del codice. È evidente che non potete fare affidamento sulla creazione di un riferimento nominativo (*named reference*) che contenga ognuno dei vostri oggetti.

```
MyType aReference;
```

dal momento che, appunto, non saprete mai di quanti oggetti potreste avere effettivamente bisogno.

La maggior parte dei linguaggi mette a disposizione tecniche specifiche per risolvere questo problema fondamentale. Java offre diversi meccanismi per contenere gli oggetti, o meglio, i riferimenti agli oggetti: il tipo supportato dal compilatore è l'array, che già conoscete. Un array rappresenta il modo più efficiente per contenere un gruppo di oggetti, ed è anche la scelta praticamente obbligata per

gestire un gruppo di primitivi. Tuttavia l'array ha dimensioni fisse: nel caso più generale, nel momento in cui scrivete il programma non sapete di quanti oggetti avrete bisogno o se dovete ricorrere a una tecnica più elaborata per memorizzare i vostri oggetti, pertanto i limiti rappresentati dalle dimensioni fisse degli array risulteranno troppo vincolanti.

La libreria **java.util** offre un insieme pressoché completo di *classi contenitore* per risolvere questo problema, di cui i tipi di base sono **List**, **Set**, **Queue** e **Map**.¹

Questi tipi di oggetto sono chiamati anche *classi collezione*, ma tenuto conto che la libreria Java adotta il nome **Collection** per indicare un determinato sottoinsieme della libreria si è scelto di utilizzare il termine *contenitore*, più completo. I contenitori mettono a disposizione tecniche ricercate per gestire gli oggetti, e vi consentiranno di risolvere un numero sorprendente di problemi.

Tra le molteplici caratteristiche dei contenitori è degno di nota il fatto che le classi contenitore di Java si ridimensionano automaticamente: a differenza degli array, quindi permettono di inserire un numero di oggetti arbitrario senza dover definire le dimensioni del contenitore nel momento in cui scrivete il programma. A differenza di altri linguaggi quali Perl, Python e Ruby, Java non supporta direttamente la creazione delle classi contenitore mediante parole chiave, tuttavia queste classi rimangono comunque strumenti fondamentali che incrementeranno in modo notevole le vostre abilità di programmatore. Questo capitolo vi fornirà una conoscenza pratica di base della libreria di contenitori Java, esaminando situazioni di utilizzo tipiche ed evidenziando i contenitori che impiegherete nella programmazione quotidiana; nel Volume 2, il Capitolo 5 tratterà i rimanenti contenitori e vi fornirà maggiori dettagli sulle loro funzionalità e modalità di utilizzo.

Generici e contenitori type-safe

Nelle versioni precedenti Java SE5, uno dei problemi dell'utilizzo dei contenitori era che il compilatore permetteva di inserire al loro interno un tipo errato. Considerate per esempio un contenitore di oggetti **Apple** che utilizza l'infaticabile contenitore di base **ArrayList**: per il momento potete considerare **ArrayList** come "un array che si estende automaticamente". L'uso di **ArrayList** è molto semplice: lo create, vi inserite oggetti con **add()** e accedete

1. **Set** contiene un insieme di elementi univoci e **Map** è un array associativo che permette di combinare gli oggetti tra loro.

con **get()** usando un indice, esattamente come fareste con un array ma senza le parentesi quadre.²

ArrayList dispone anche del metodo **size()** per sapere di quanti elementi è composto l'array, in modo da evitare l'involontario superamento dell'indice: questo evento darebbe infatti luogo a un'eccezione, un argomento che sarà trattato nel Volume 2, Capitolo 1.

In questo esempio nel contenitore vengono inseriti oggetti **Apple** e **Orange**, poi estratti. Di norma il compilatore Java visualizzerebbe un avvertimento poiché l'esempio non utilizza i generici, tuttavia in questo caso particolare si è fatto uso di un'annotazione Java SE5 allo scopo di eliminare tale avvertimento.

Le annotazioni iniziano con il simbolo @ e accettano un argomento: nel caso dell'annotazione **@SuppressWarnings** l'argomento indica che dovranno essere eliminati soltanto gli avvertimenti (*warning*) di tipo "unchecked".

```
//: holding/ApplesAndOrangesWithoutGenerics.java
// Semplice esempio di contenitore (il compilatore visualizza
// avvertimenti).
// {ThrowsException}
import java.util.*;

class Apple {
    private static long counter;
    private final long id = counter++;
    public long id() { return id; }
}

class Orange {}

public class ApplesAndOrangesWithoutGenerics {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        ArrayList apples = new ArrayList();
        for(int i = 0; i < 3; i++)
            apples.add(new Apple());
        System.out.println(apples.get(0).id());
        System.out.println(apples.get(1).id());
        System.out.println(apples.get(2).id());
    }
}
```

2. L'**ArrayList** è uno di quei casi in cui sarebbe stato utile poter eseguire l'overloading degli operatori, che nelle classi contenitore C++ e C# ha semplificato la sintassi.

```

    apples.add(new Apple());
    // Non viene impedito di aggiungere una Orange alle Apple:
    apples.add(new Orange());
    for(int i = 0; i < apples.size(); i++)
        ((Apple)apples.get(i)).id();
    // Orange viene rilevato soltanto in fase di esecuzione
}
} /* (Da eseguire per visualizzare l'output) */:~

```

Le annotazioni saranno trattate in dettaglio nel Volume 2, Capitolo 8.

Apple e **Orange** sono due classi distinte che non hanno niente in comune tranne il fatto che sono entrambe **Object**: ricordate che se non specificate il nome della classe da cui ereditate, Java considererà automaticamente **Object**. Dal momento che l'**ArrayList** contiene alcuni **Object**, a questo contenitore è possibile non soltanto aggiungere oggetti **Apple** utilizzando il metodo **add()** di **ArrayList**, ma anche includervi oggetti **Orange** senza problemi, né in fase di compilazione né al momento dell'esecuzione. Quando recupererete quelli che ritenete essere oggetti **Apple** con il **get()** di **ArrayList**, Java restituirà un riferimento a **Object** che dovrete poi sottoporre a cast ad **Apple**. È quindi indispensabile racchiudere l'intera espressione tra parentesi per forzare la valutazione del cast prima di chiamare il metodo **id()** per **Apple**: in caso contrario si otterebbe un errore di sintassi.

Al momento dell'esecuzione, quando proverete a eseguire il cast dell'oggetto **Orange** ad **Apple** otterrete un errore visualizzato sotto forma dell'eccezione già menzionata.

Nel Volume 2, Capitolo 3, vedrete che la creazione di classi utilizzando i generici di Java può rivelarsi complessa. In ogni caso, l'applicazione di classi generiche predefinite è solitamente un'operazione più immediata. Per esempio, allo scopo di definire un **ArrayList** che debba contenere oggetti **Apple** basterà scrivere **ArrayList<Apple>** invece di **ArrayList**.

Notate i simboli di minore e maggiore (< e >) che includono il parametro del tipo, anche più di uno: questi parametri specificano il tipo o i tipi che possono essere gestiti dall'istanza del contenitore.

Con i generici, in fase di compilazione è impossibile inserire in un contenitore il tipo di oggetto errato.³

3. Al termine del Capitolo 3 viene presa in esame l'eventualità che questo aspetto debba essere considerato negativamente. Nello stesso capitolo scoprirete anche che i generici Java non sono utili soltanto per i contenitori type-safe.

Di seguito è riproposto l'esempio, utilizzando i generici.

```

//: holding/ApplesAndOrangesWithGenerics.java
import java.util.*;

public class ApplesAndOrangesWithGenerics {
    public static void main(String[] args) {
        ArrayList<Apple> apples = new ArrayList<Apple>();
        for(int i = 0; i < 3; i++)
            apples.add(new Apple());
        // Errore di compilazione:
        // apples.add(new Orange());
        for(int i = 0; i < apples.size(); i++)
            System.out.println(apples.get(i).id());
        // Uso di foreach:
        for(Apple c : apples)
            System.out.println(c.id());
    }
} /* Output:
0
1
2
0
1
2
*///:~

```

Ora il compilatore vi impedirà di inserire un oggetto **Orange** tra quelli **Apple**, generando così un errore di compilazione invece che di esecuzione.

Notate anche che nell'ottenimento degli elementi da **List** il casting non è più necessario; poiché **List** conosce il tipo del suo contenuto, esegue automaticamente il cast quando chiamate **get()**.

Pertanto, i generici non soltanto danno la garanzia che il compilatore controlli il tipo di oggetto inserito in un contenitore, ma mettono a disposizione una sintassi più lineare per l'utilizzo degli elementi che vi sono contenuti.



L'esempio mostra anche che, se non avete bisogno di utilizzare l'indice di ogni elemento, potete servirvi della sintassi `foreach` per selezionare gli elementi in `List`.

Quando specificate un tipo come parametro generico, Java non vi costringe a collocare nel contenitore soltanto quel tipo esatto di oggetto; infatti l'upcasting funziona anche con i generici, esattamente come con altri tipi di dato.

```
//: holding/GenericsAndUpcasting.java
import java.util.*;

class GrannySmith extends Apple {}
class Gala extends Apple {}
class Fuji extends Apple {}
class Braeburn extends Apple {}

public class GenericsAndUpcasting {
    public static void main(String[] args) {
        ArrayList<Apple> apples = new ArrayList<Apple>();
        apples.add(new GrannySmith());
        apples.add(new Gala());
        apples.add(new Fuji());
        apples.add(new Braeburn());
        for(Apple c : apples)
            System.out.println(c);
    }
} /* Output: (Esempio)
GrannySmith@7d772e
Gala@11b86e7
Fuji@35ce36
Braeburn@757aef
*///:~
```

Potete quindi aggiungere un sottotipo di `Apple` a un contenitore che è previsto contenere oggetti `Apple`.

L'output viene prodotto dal metodo predefinito `toString()` di `Object`, che visualizza il nome della classe seguito dalla rappresentazione esadecimale



senza segno del codice hash dell'oggetto, generato dal metodo `hashCode()`; questo argomento sarà esaminato in dettaglio nel Volume 2, Capitolo 5.

Esercizio 1 (2) Create una nuova classe chiamata `Gerbil` con `int gerbilNumber` che viene inizializzato nel costruttore; assegnate alla classe stessa un metodo `hop()` che visualizza il numero corrispondente al gerbillo che in quel momento sta saltellando (`hop`). Create un `ArrayList` e aggiungete oggetti `Gerbil` a `List`, poi utilizzate il metodo `get()` per scorrere `List` e chiamate `hop()` per ogni gerbillo.

Concetti di base

La libreria di contenitori Java scinde il problema di “gestione degli oggetti” in due concetti distinti, espressi come interfacce di base della libreria.

1. Collection (collezione): una raccolta sequenziale di singoli elementi ai quali sono applicate una o più regole. Una `List` deve conservare gli elementi nello stesso ordine in cui sono stati inseriti, un `Set` non può contenere elementi duplicati e una `Queue` produce gli elementi nell'ordine determinato da una disciplina di accodamento (*queuing discipline*), corrispondente di norma all'ordine di inserimento.

2. Map (mappa): un gruppo di coppie chiave-valore indicanti gli oggetti, che permettono di recuperare un valore mediante la chiave a esso associata. In un `ArrayList` è possibile cercare un oggetto utilizzando un numero, cosicché ha perfettamente senso associare numeri agli oggetti. Una mappa consente di cercare un oggetto utilizzando un altro oggetto: è chiamata anche *array associativo*, poiché associa oggetti ad altri oggetti, o *dizionario*, poiché permette di recuperare un valore tramite un oggetto-chiave, esattamente come in un dizionario si può verificare una definizione cercando un vocabolo.

Le `Map` sono strumenti di programmazione molto potenti.

Benché non sia sempre possibile, in teoria potreste scrivere la maggior parte del vostro codice in modo che “colloqui” con queste interfacce, e l'unico punto in cui dovreste specificare il tipo esatto è al momento della creazione del contenitore. Sarebbe quindi possibile creare una `List` come la seguente:

```
List<Apple> apples = new ArrayList<Apple>();
```

Notate che l'`ArrayList` viene sottoposto a upcast a `List`, in modo diverso da come è stato gestito negli esempi precedenti. L'utilità dell'interfaccia diventa evidente quando avete bisogno di variare la vostra implementazione; a quel



punto non dovete fare altro che modificarla nel momento della sua creazione, come in:

```
List<Apple> apples = new LinkedList<Apple>();
```

Per fare questo generalmente creerete un oggetto di una classe concreta, lo sottoporrete a upcast all'interfaccia corrispondente e vi servirete dell'interfaccia nel resto del codice. Tale approccio, tuttavia, non funzionerà sempre poiché alcune classi dispongono di funzionalità aggiuntive. Per esempio **LinkedList** ha metodi che non sono disponibili nell'interfaccia **List**, e **TreeMap** ha metodi che non si trovano nell'interfaccia **Map**. Se avete bisogno di utilizzarli non sarete in grado di eseguire l'upcast all'interfaccia più generica.

L'interfaccia **Collection** generalizza l'idea di *sequenza*, ossia un modo per gestire un gruppo di oggetti. Di seguito troverete un semplice esempio che popola una **Collection** di oggetti **Integer**, rappresentata in questo caso da un **ArrayList**, poi visualizza ogni elemento del contenitore risultante.

```
//: holding/SimpleCollection.java
import java.util.*;

public class SimpleCollection {
    public static void main(String[] args) {
        Collection<Integer> c = new ArrayList<Integer>();
        for(int i = 0; i < 10; i++)
            c.add(i); // Autoboxing
        for(Integer i : c)
            System.out.print(i + ", ");
    }
} /* Output:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
*//*:~
```

Poiché in questo codice vengono utilizzati soltanto metodi di **Collection**, qualsiasi oggetto di una classe ereditata da **Collection** funzionerebbe ugualmente; in ogni caso **ArrayList** è il tipo di sequenza più comunemente utilizzato.

Il nome del metodo **add()** suggerisce che la sua funzione è inserire un nuovo elemento nella collezione. Tenete presente, però, che la documentazione Java definisce esattamente che **add()** "assicura che la **Collection** corrente conten-



ga l'elemento specificato": questa precisione lessicale si è resa necessaria per supportare il significato di **Set**, che aggiunge l'elemento soltanto se non esiste già. Con un **ArrayList** o con qualsiasi tipo di **List**, **add()** significa sempre "inserire", poiché questi contenitori non tengono conto della presenza di duplicati.

Tutte le **Collection** possono essere iterate mediante la sintassi `foreach`, come mostra l'esempio precedente; nel prosieguo del capitolo verrete a conoscenza di una tecnica più flessibile che utilizza il cosiddetto *iteratore*.

Esercizio 2 (1) Modificate **SimpleCollection.java** per utilizzare un **Set** per **c**.

Esercizio 3 (2) Modificate **innerclasses/Sequence.java** in modo da potervi aggiungere qualsiasi numero di elementi.

Aggiunta di gruppi di elementi

In **java.util** le classi **Arrays** e **Collections** offrono metodi pratici per aggiungere gruppi di elementi a una collezione. **Arrays.asList()** accetta un array o un elenco di elementi separati da virgolette (utilizzando elenchi di argomenti variabili, o *varargs*) e li trasforma in un oggetto **List**. **Collections.addAll()** accetta come parametri un oggetto **Collection** e un array o un elenco di elementi separati da virgolette e aggiunge gli elementi alla **Collection** specificata.

Ecco un esempio che illustra entrambe le tecniche, nonché il metodo più convenzionale **addAll()** che è disponibile in tutti i tipi di **Collection**.

```
//: holding/AddingGroups.java
// Aggiunta di gruppi di elementi agli oggetti Collection.
import java.util.*;

public class AddingGroups {
    public static void main(String[] args) {
        Collection<Integer> collection =
            new ArrayList<Integer>(Arrays.asList(1, 2, 3, 4, 5));
        Integer[] moreInts = { 6, 7, 8, 9, 10 };
        collection.addAll(Arrays.asList(moreInts));
        // Offre prestazioni di gran lunga superiori,
        // ma non permette di costruire una Collection:
        Collections.addAll(collection, 11, 12, 13, 14, 15);
    }
}
```



```

    Collections.addAll(collection, moreInts);
    // Produce un elenco sostenuto da un array:
    List<Integer> list = Arrays.asList(16, 17, 18, 19, 20);
    list.set(1, 99); // OK - modifica un elemento
    // list.add(21); // Errore di runtime, perche' l'array
                    // sottostante non puo' essere
                    // ridimensionato.
}
} //:~
```

Il costruttore di una **Collection** può accettare un'altra **Collection** di cui si serve in fase di inizializzazione, al fine di consentirvi di utilizzare **Arrays.asList()** per produrre l'input per il costruttore. Tuttavia **Collections.addAll()** viene eseguito più rapidamente e permette di costruire **Collection** senza elementi e di chiamare poi **Collections.addAll()**, pertanto è considerato preferibile.

Di contro il metodo membro **Collection.addAll()** che accetta come argomento soltanto un altro oggetto **Collection** non è flessibile quanto **Arrays.asList()** o **Collections.addAll()**, che utilizzano elenchi di argomenti variabili.

Potete anche utilizzare l'output di **Arrays.asList()** direttamente, come in **List**, ma in tal caso la rappresentazione sottostante sarà l'array, che non è ridimensionabile. Se in questo oggetto cercate di aggiungere (**add()**) o cancellare (**delete()**) elementi, poiché queste operazioni modificano la dimensione di un array otterrete un errore di tipo *Unsupported Operation* in fase di esecuzione.

Uno dei limiti di **Arrays.asList()** è che cerca di determinare al meglio il tipo di **List**, senza curarsi di ciò che gli assegnate. Questo talvolta può causare problemi.

```

//: holding/AsListInference.java
// Arrays.asList() cerca di determinare il tipo.
import java.util.*;

class Snow {}
class Powder extends Snow {}
class Light extends Powder {}
class Heavy extends Powder {}
```

```

class Crusty extends Snow {}
class Slush extends Snow {}

public class AsListInference {
    public static void main(String[] args) {
        List<Snow> snow1 = Arrays.asList(
            new Crusty(), new Slush(), new Powder());

        // Non compila:
        // List<Snow> snow2 = Arrays.asList(
        //     new Light(), new Heavy());
        // Messaggio del compilatore:
        // found   : java.util.List<Powder>
        // required: java.util.List<Snow>

        // Collections.addAll() non si confonde:
        List<Snow> snow3 = new ArrayList<Snow>();
        Collections.addAll(snow3, new Light(), new Heavy());

        // Fornisce un suggerimento specificando
        // esplicitamente il tipo di argomento:
        List<Snow> snow4 = Arrays.<Snow>asList(
            new Light(), new Heavy());
    }
} //:~
```

Quando prova a creare **snow2**, **Arrays.asList()** ha soltanto tipi di **Powder**, di conseguenza crea una **List<Powder>** invece di una **List<Snow>**; di contro **Collections.addAll()** non dà problemi poiché è in grado di determinare dal primo argomento il tipo di destinazione.

Come potete notare dalla creazione di **snow4** è possibile inserire un “suggerimento” all'interno di **Arrays.asList()**, per indicare al compilatore quale deve essere il tipo di destinazione effettivo per la **List** prodotta da **Arrays.asList()**: questa indicazione è nota come *specifica esplicita del tipo di argomento (explicit type argument specification)*.

Come vedrete le **Map** sono più complesse e la libreria standard Java non fornisce alcun metodo di inizializzazione automatica, tranne che mediante il contenuto di un'altra mappa.



Visualizzazione e stampa

Per produrre una rappresentazione visualizzabile o stampabile di un array è necessario ricorrere ad `Arrays.toString()`, tuttavia la visualizzazione o la stampa dei contenitori non presenta alcun problema. Di seguito è mostrato un esempio che illustra anche l'utilizzo dei contenitori di base.

```
//: holding/PrintingContainers.java
// I contenitori si visualizzano o stampano automaticamente.
import java.util.*;
import static net.mindview.util.Print.*;

public class PrintingContainers {
    static Collection fill(Collection<String> collection) {
        collection.add("rat");
        collection.add("cat");
        collection.add("dog");
        collection.add("dog");
        return collection;
    }
    static Map fill(Map<String, String> map) {
        map.put("rat", "Fuzzy");
        map.put("cat", "Rags");
        map.put("dog", "Bosco");
        map.put("dog", "Spot");
        return map;
    }
    public static void main(String[] args) {
        print(fill(new ArrayList<String>()));
        print(fill(new LinkedList<String>()));
        print(fill(new HashSet<String>()));
        print(fill(new TreeSet<String>()));
        print(fill(new LinkedHashSet<String>()));
        print(fill(new HashMap<String, String>()));
        print(fill(new TreeMap<String, String>()));
        print(fill(new LinkedHashMap<String, String>()));
    }
}
```

```
} /* Output:
[rat, cat, dog, dog]
[rat, cat, dog, dog]
[rat, dog, cat]
[dog, cat, rat]
[rat, cat, dog]
{rat=Fuzzy, dog=Spot, cat=Rags}
{dog=Spot, cat=Rags, rat=Fuzzy}
{rat=Fuzzy, cat=Rags, dog=Spot}
*///:~
```

Questo codice evidenzia le due categorie principali disponibili nella libreria dei contenitori Java, la cui distinzione si basa sul numero di elementi che si trovano in ogni “alloggiamento” nel contenitore. La categoria **Collection** contiene un solo elemento in ogni alloggiamento; essa include **List**, che mantiene un gruppo di elementi nella sequenza specificata, **Set**, che permette l’aggiunta di un solo elemento identico, e **Queue**, che permette di inserire gli oggetti a un “estremità” del contenitore e di rimuoverli dall’altra “estremità”: tenete presente che, per gli scopi che si prefigge questo esempio, è solo un modo diverso di osservare una sequenza, pertanto non viene esaminato. Una **Map** contiene due oggetti in ogni alloggiamento, una chiave e il valore a essa associato.

Dall’output potete notare che il comportamento predefinito di visualizzazione, fornito dal metodo `toString()` di ogni contenitore, produce risultati accettabili. Una **Collection** viene visualizzata tra parentesi quadre, nelle quali ogni elemento è separato da una virgola; una **Map** è inclusa tra parentesi graffe: ogni coppia di chiave/valore è associata al simbolo di uguale (=), con le chiavi a sinistra e i valori a destra.

Il primo metodo `fill()` funziona con tutti i tipi di **Collection**, ciascuno dei quali implementa il metodo `add()` per includere nuovi elementi.

ArrayList e **LinkedList** sono due tipi di **List** e come potete notare dall’output entrambi conservano gli elementi nello stesso ordine in cui sono stati inseriti; la differenza tra i due non consiste soltanto nelle prestazioni di alcune operazioni, ma anche nel fatto che una **LinkedList** può contenere più elementi rispetto a un **ArrayList**. Avrete modo di approfondire questi aspetti nel proseguito del capitolo.

HashSet, **TreeSet** e **LinkedHashSet** sono tipi di **Set**. L’output mette in risalto che un **Set** contiene soltanto elementi univoci, e che varie implementazioni di **Set** memorizzano gli elementi in modo diverso. L’**HashSet** memorizza gli ele-



menti ricorrendo a un approccio alquanto complesso che vedrete nel Volume 2, Capitolo 5; per il momento vi basterà sapere che questa tecnica è il modo più veloce per recuperare elementi, sebbene l'ordine in cui essi vengono memorizzati possa apparire privo di senso: spesso è infatti sufficiente sapere se un **Set** contiene un determinato elemento, non l'ordine in cui viene visualizzato. Qualora l'ordine di memorizzazione sia importante potrete servirvi di un **TreeSet**, che mantiene gli oggetti in ordine di confronto ascendente, o di un **LinkedHashSet**, che conserva gli oggetti nell'ordine in cui sono stati aggiunti.

Una **Map** (array associativo) consente di recuperare un oggetto tramite una chiave, come se si trattasse di un semplice database: l'oggetto associato è chiamato valore. Se avete una **Map** che associa gli stati alle loro capitali e volete conoscere la capitale della Lituania, eseguirete una ricerca utilizzando la chiave "Lituania", come se lavoraste con l'indice di un array. A causa di questo comportamento **Map** accetta soltanto una chiave per volta.

Map.put(key, value) aggiunge un valore (il risultato da ottenere) e lo associa a una chiave (l'elemento da ricercare); **Map.get(key)** fornisce il valore associato a una determinata chiave. Tenete presente che il codice dell'esempio precedente si limita ad aggiungere coppie di chiave/valore senza eseguire alcuna ricerca: questo è un argomento che vedrete in seguito.

Ricordate che non dovete specificare né preoccuparvi delle dimensioni di **Map**, che si ridimensiona in modo automatico; inoltre le mappe sono perfettamente visualizzabili e mostrano l'associazione tra chiavi e valori. L'ordine in cui chiavi e valori sono contenuti in **Map** non è quello di inserimento, poiché l'implementazione **HashMap** utilizza un algoritmo molto efficiente che controlla questo ordinamento.

L'esempio utilizza le tre varianti di base di **Map**: **HashMap**, **TreeMap** e **LinkedHashMap**. Come **HashSet**, anche **HashMap** fornisce la tecnica di ricerca più veloce e non conserva i propri elementi in nessun ordine evidente.

TreeMap mantiene le chiavi in ordine di confronto ascendente, mentre **LinkedHashMap** utilizza l'ordine di inserimento pur garantendo prestazioni di ricerca analoghe a quelle di **HashMap**.

Esercizio 4 (3) Create una classe generatore che visualizzi, sotto forma di oggetti **String**, i nomi dei personaggi del vostro film preferito ogni volta che chiamate **next()** e riprenda nuovamente dall'inizio dell'elenco dopo aver visualizzato l'ultimo nome. Utilizzate questo generatore per popolare un array, un **ArrayList**, una **LinkedList**, un **HashSet**, un **LinkedHashSet** e un **TreeSet**, poi visualizzate tutti i contenitori.



List

Gli oggetti **List** consentono di mantenere gli elementi in una determinata sequenza. L'interfaccia **List** integra **Collection** con numerosi metodi che permettono di inserire e rimuovere gli elementi all'interno di una **List**. Esistono due tipi di **List**:

1. L'**ArrayList** di base, molto efficiente nell'accesso casuale agli elementi, tuttavia più lento nelle operazioni di inserimento e cancellazione degli elementi da una **List**.
2. **LinkedList**, che garantisce un accesso sequenziale ottimale, con inserimenti e cancellazioni non particolarmente onerose.

La **LinkedList** è relativamente lenta nell'accesso casuale, ma offre un numero maggiore di caratteristiche rispetto all'**ArrayList**. Il seguente esempio anticipa il Volume 2, Capitolo 2 e utilizza la libreria **typeinfo.pets**, che mette a disposizione una gerarchia di classi **Pet** e alcuni strumenti per generare casualmente oggetti **Pet**. Per il momento non avete necessità di conoscere tutti i dettagli, ma vi basterà sapere che la libreria contiene una classe **Pet** e alcuni sottotipi di **Pet** e il metodo **static Pets.arrayList()** restituisce un'**ArrayList** popolata di oggetti **Pet** selezionati casualmente.

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

public class ListFeatures {
    public static void main(String[] args) {
        Random rand = new Random(47);
        List<Pet> pets = Pets.arrayList(7);
        print("1: " + pets);
        Hamster h = new Hamster();
        pets.add(h); // Ridimensiona automaticamente
        print("2: " + pets);
        print("3: " + pets.contains(h));
        pets.remove(h); // Rimuove tramite l'oggetto
        Pet p = pets.get(2);
        print("4: " + p + " " + pets.indexOf(p));
        Pet cymric = new Cymric();
        print("5: " + pets.indexOf(cymric));
```



```
print("6: " + pets.remove(cymric));
// Deve essere l'oggetto esatto:
print("7: " + pets.remove(p));
print("8: " + pets);
pets.add(3, new Mouse()); // Inserisce un indice
print("9: " + pets);
List<Pet> sub = pets.subList(1, 4);
print("subList: " + sub);
print("10: " + pets.containsAll(sub));
Collections.sort(sub); // Ordinamento istantaneo
print("sorted subList: " + sub);
// L'ordine e' irrilevante in containsAll():
print("11: " + pets.containsAll(sub));
Collections.shuffle(sub, rand); // Mescola
print("shuffled subList: " + sub);
print("12: " + pets.containsAll(sub));
List<Pet> copy = new ArrayList<Pet>(pets);
sub = Arrays.asList(pets.get(1), pets.get(4));
print("sub: " + sub);
copy.retainAll(sub);
print("13: " + copy);
copy = new ArrayList<Pet>(pets); // Ottiene una nuova
// copia
copy.remove(2); // Rimuove tramite indice
print("14: " + copy);
copy.removeAll(sub); // Rimuove soltanto gli oggetti
// identici
print("15: " + copy);
copy.set(1, new Mouse()); // Sostituisce un elemento
print("16: " + copy);
copy.addAll(2, sub); // Inserisce un elenco al centro
print("17: " + copy);
print("18: " + pets.isEmpty());
pets.clear(); // Rimuove tutti gli elementi
print("19: " + pets);
print("20: " + pets.isEmpty());
```



```
pets.addAll(Pets.arrayList(4));
print("21: " + pets);
Object[] o = pets.toArray();
print("22: " + o[3]);
Pet[] pa = pets.toArray(new Pet[0]);
print("23: " + pa[3].id());
}
} /* Output:
1: [Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug]
2: [Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug, Hamster]
3: true
4: Cymric 2
5: -1
6: false
7: true
8: [Rat, Manx, Mutt, Pug, Cymric, Pug]
9: [Rat, Manx, Mutt, Mouse, Pug, Cymric, Pug]
subList: [Manx, Mutt, Mouse]
10: true
sorted subList: [Manx, Mouse, Mutt]
11: true
shuffled subList: [Mouse, Manx, Mutt]
12: true
sub: [Mouse, Pug]
13: [Mouse, Pug]
14: [Rat, Mouse, Mutt, Pug, Cymric, Pug]
15: [Rat, Mutt, Cymric, Pug]
16: [Rat, Mouse, Cymric, Pug]
17: [Rat, Mouse, Mouse, Pug, Cymric, Pug]
18: false
19: []
20: true
21: [Manx, Cymric, Rat, EgyptianMau]
22: EgyptianMau
23: 14
*///:~
```



Le righe di output sono numerate per essere messe in relazione con il codice sorgente. La prima riga visualizza la **List** di **Pet** originale. A differenza di un array, una **List** permette di aggiungere o rimuovere elementi dopo la sua creazione ed è ridimensionabile: il suo pregio fondamentale è appunto quello di essere una sequenza modificabile. Potete vedere il risultato dell'aggiunta di un **Hamster** nella riga 2, a dimostrazione che l'oggetto viene accodato all'elenco.

Per determinare se un oggetto è nella **List** si utilizza il metodo **contains()**, mentre per rimuovere un oggetto basterà passarne il riferimento al metodo **remove()**. Inoltre, come potete vedere nella riga 4, disponendo di un riferimento a un oggetto è possibile determinare il numero indice che corrisponde alla posizione dell'oggetto nell'elenco, tramite il metodo **indexOf()**.

Per comprendere se un elemento è contenuto in una **List**, eventualmente allo scopo di scoprirne l'indice e rimuoverlo, si può utilizzare il metodo **equals()** che è parte della classe radice **Object**.

Ogni oggetto **Pet** è definito per essere univoco: sebbene nell'elenco esistano due **Cymric**, quindi, se create un nuovo oggetto **Cymric** e lo passate a **indexOf()** otterrete il risultato **-1**, indicante che l'oggetto non è stato trovato, e se tentate di rimuovere (**remove()**) l'oggetto il programma restituirà **false**. Per altre classi il metodo **equals()** può essere definito in modo diverso: per esempio, due **String** sono considerate uguali se il loro contenuto è identico. Di conseguenza, per evitare sorprese, è importante sapere che il comportamento di **List** varia in funzione del comportamento di **equals()**.

Le righe 7 e 8 evidenziano che l'eliminazione di un oggetto esattamente corrispondente a un oggetto presente in **List** ha avuto successo.

Come potete vedere nella riga 9 e nel codice che la precede è possibile inserire un elemento in qualsiasi punto della **List**, tuttavia questo dà luogo a un problema: per una **LinkedList**, inserimento e rimozione nel mezzo di un elenco sono operazioni di routine, ma per un **ArrayList** si tratta di attività assai onerose. Questo non significa certo che non dobbiate eseguire inserimenti nel mezzo di un **ArrayList**, preferendogli invece sempre una **LinkedList**; dovete tuttavia prendere atto del fatto che, in caso di grandi quantità di inserimenti in un **ArrayList**, qualora il vostro programma iniziasse a rallentare dovreste considerare la vostra implementazione di **List** come possibile colpevole. Tenete presente che una delle migliori tecniche per scoprire questi colli di bottiglia è l'impiego di un *profiler*, come vedrete nel supplemento disponibile all'indirizzo <http://mindview.net/Books/BetterJava>.

L'ottimizzazione è un argomento complesso e l'orientamento migliore è non tenerne conto finché le prestazioni non vi creino effettivamente degli inconvenienti: in ogni caso, comprendere i problemi è sempre un buon approccio.



Il metodo **subList()** permette di ritagliare una "porzione" da un elenco di grandi dimensioni, e questo naturalmente fornisce un risultato **true**, una volta passato a **containsAll()** con il parametro dell'elenco più grande. Notate anche come l'ordine sia ininfluente: nelle righe 11 e 12 risulta chiaro che le chiamate **Collections.sort()** e **Collections.shuffle()** su **sub** non hanno effetto sul risultato di **containsAll()**; **subList()** elabora un elenco "sostenuto" dalla lista originale. Di conseguenza i cambiamenti nell'elenco restituito corrisponderanno alla lista originale e viceversa.

Il metodo **retainAll()** è effettivamente un'operazione di "intersezione di insiemi", che in questo caso conserva tutti gli elementi che si trovano sia in **copy** sia in **sub**. Anche in questo caso il comportamento risultante dipende dal metodo **equals()**.

La riga 14 mostra il risultato della rimozione di un elemento mediante il suo numero indice, una tecnica più semplice rispetto all'utilizzo del riferimento all'oggetto perché con gli indici non occorre tenere conto del comportamento degli **equals()**.

Anche il metodo **removeAll()** opera basandosi sul metodo **equals()**: come indica il suo nome, rimuove tutti gli oggetti dalla **List** che sono specificati nell'argomento **List**.

I progettisti Java, purtroppo, hanno scelto un nome infelice per il metodo **set()**, che potrebbe generare confusione con la classe **Set**: "replace" sarebbe stato un nome migliore, considerato che il metodo sostituisce l'elemento corrispondente all'indice specificato come primo argomento, con l'oggetto indicato come secondo argomento.

La riga 17 mostra che **List** possiede un metodo **addAll()** sovraccarico mediante il quale potete inserire la nuova lista in qualsiasi punto di quello originale, invece di accodarla con il metodo **addAll()** di **Collection**.

Nelle righe da 18 a 20 è visualizzato l'effetto dei metodi **isEmpty()** e **clear()**.

Le righe 22 e 23 indicano come convertire qualsiasi **Collection** in un array, ricorrendo al metodo **toArray()**. Si tratta di un metodo sovraccarico: la versione senza argomenti restituisce un array di **Object**, ma se passate un array del tipo di destinazione alla versione sovraccarica ne otterrete uno del tipo specifico, sempre che questo sia coerente. Se l'array di argomenti è troppo limitato per contenere tutti gli oggetti in **List**, come in questo caso, **toArray()** creerà un nuovo array della dimensione appropriata. Gli oggetti **Pet** hanno un metodo **id()**, che potete vedere chiamato su uno degli oggetti nell'array risultante.

Esercizio 5 (3) Modificate **ListFeatures.java** in modo che utilizzi valori **Integer** invece di **Pet** (ricordatevi dell'autoboxing!), e motivate le differenze nei risultati.

Esercizio 6 (2) Modificate **ListFeatures.java** in modo che utilizzi **String** invece di **Pet** e motivate qualsiasi differenza nei risultati.

Esercizio 7 (3) Create una classe, poi un array inizializzato di oggetti della vostra classe. Popolate una **List** con il contenuto del vostro array. Create un sottoinsieme di **List** utilizzando **subList()**, quindi rimuovete questo sottoinsieme dalla **List**.

Iterator

In qualsiasi contenitore deve essere disponibile un meccanismo per inserire e ottenere gli elementi: dopotutto, l'obiettivo primario di un contenitore è appunto quello di contenere oggetti. In una **List**, **add()** è uno dei metodi per inserire elementi, e **get()** un metodo per recuperarli.

Quando volete iniziare a ragionare in termini più astratti, però, si presenta un inconveniente: programmando dovete tenere conto del tipo esatto di contenitore da utilizzare. Questo a prima vista potrebbe non sembrare un ostacolo insormontabile, tuttavia lo diventa se scrivete codice per una **List** e in seguito scoprite che sarebbe più opportuno applicare lo stesso codice a un **Set**; oppure fin dall'inizio potreste desiderare scrivere codice che non tenga conto del tipo di contenitore con cui dovrà interfacciarsi, per poterlo utilizzare su vari tipi di contenitori senza bisogno di riscrivere.

Per ottenere questo livello di astrazione è possibile ricorrere al design pattern **Iterator** (iteratore), un'altra struttura di progettazione. Un iteratore è un oggetto incaricato di spostarsi all'interno di una sequenza per selezionare ogni oggetto che vi sia contenuto, senza che il programmatore client sia tenuto a occuparsi della struttura sottostante.

Un iteratore è chiamato anche *lightweight object*, letteralmente “oggetto leggero”, per la praticità con cui può essere creato, ma esistono alcune limitazioni degli iteratori Java, per esempio il fatto che possono spostarsi in una sola direzione.

Le uniche operazioni che è possibile eseguire con un **Iterator** sono:

1. ordinare a una **Collection** di passare un **Iterator**, tramite l'omonimo metodo **iterator()**: così facendo l'**Iterator** sarà pronto a restituire il primo elemento nella sequenza;
2. ottenere l'oggetto successivo nella sequenza, con **next()**;
3. verificare se la sequenza contiene altri oggetti, con **hasNext()**;
4. rimuovere l'ultimo elemento restituito dall'iteratore, con **remove()**.

Per vedere all'opera l'iteratore utilizzerete l'utility **Pet** del Volume 2, Capitolo 2 (**typeinfo.pets**).

```
//: holding/SimpleIteration.java
import typeinfo.pets.*;
import java.util.*;

public class SimpleIteration {
    public static void main(String[] args) {
        List<Pet> pets = Pets.arrayList(12);
        Iterator<Pet> it = pets.iterator();
        while(it.hasNext()) {
            Pet p = it.next();
            System.out.print(p.id() + ":" + p + " ");
        }
        System.out.println();
        // Un approccio piu' semplice, quando possibile:
        for(Pet p : pets)
            System.out.print(p.id() + ":" + p + " ");
        System.out.println();
        // Un Iterator puo' anche rimuovere elementi:
        it = pets.iterator();
        for(int i = 0; i < 6; i++) {
            it.next();
            it.remove();
        }
        System.out.println(pets);
    }
} /* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx 8:
Cymric 9:Rat 10:EgyptianMau 11:Hamster
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx 8:
Cymric 9:Rat 10:EgyptianMau 11:Hamster
[Pug, Manx, Cymric, Rat, EgyptianMau, Hamster]
*///:~
```

Con un **Iterator** non dovete preoccuparvi del numero di elementi nel contenitore: di questo si occupano **hasNext()** e **next()**.

Come potete osservare, la sintassi `foreach` risulta più concisa quando dovete semplicemente spostarvi all'interno della **List** senza modificare l'oggetto **List** stesso.

Un **Iterator** è anche in grado di rimuovere l'ultimo elemento ottenuto da `next()`, il che significa dovere chiamare `next()` prima di `remove()`.⁴

Il concetto di prendere un contenitore di oggetti e percorrerlo al fine di eseguire un'operazione su ogni oggetto è estremamente potente e verrà analizzato in tutto il resto del volume.

Per il momento create un metodo `display()` indipendente dal contenitore.

```
//: holding/CrossContainerIteration.java
import typeinfo.pets.*;
import java.util.*;

public class CrossContainerIteration {
    public static void display(Iterator<Pet> it) {
        while(it.hasNext()) {
            Pet p = it.next();
            System.out.print(p.id() + ": " + p + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        ArrayList<Pet> pets = Pets.arrayList(8);
        LinkedList<Pet> petsLL = new LinkedList<Pet>(pets);
        HashSet<Pet> petsHS = new HashSet<Pet>(pets);
        TreeSet<Pet> petsTS = new TreeSet<Pet>(pets);
        display(pets.iterator());
        display(petsLL.iterator());
        display(petsHS.iterator());
        display(petsTS.iterator());
    }
}
```

4. `remove()` è uno dei cosiddetti “metodi opzionali” che non sono disponibili in tutte le implementazioni di **Iterator**, un argomento che vedrete meglio nel Volume 2, Capitolo 5; in ogni caso, i contenitori della libreria standard Java implementano `remove()`, pertanto non avrete alcun problema fino al Capitolo 5.

```
} /* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx 2:Cymric 0:Rat
5:Cymric 2:Cymric 7:Manx 1:Manx 3:Mutt 6:Pug 4:Pug 0:Rat
*/:-~
```

Noteate che `display()` non contiene informazioni sul tipo di sequenza in cui si trova, e questo mostra la vera forza dell'**Iterator**: la capacità di separare l'operazione di attraversamento di una sequenza dalla struttura sottostante.

Per questo motivo talvolta si suole dire che gli iteratori unificano l'accesso a contenitori.

Esercizio 8 (1) Modificate l'Esercizio 1 in modo da utilizzare un **Iterator** per spostarsi attraverso la **List**, chiamando `hop()`.

Esercizio 9 (4) Modificate **innerclasses/Sequence.java** in modo che **Sequence** funzioni con un **Iterator** invece che con un **Selector**.

Esercizio 10 (2) Modificate l'Esercizio 9 del Capitolo 8 affinché utilizzi un'ArrayList per contenere i roditori (**Rodent**) e **Iterator** per spostarsi nella sequenza di **Rodent**.

Esercizio 11 (2) Scrivete un metodo che utilizza un **Iterator** per procedere passo passo all'interno di una **Collection** e visualizza il risultato di `toString()` per ogni oggetto nel contenitore. Popolate tutti i diversi tipi di **Collection** con alcuni oggetti e applicate il vostro metodo a ogni contenitore.

ListIterator

Il **ListIterator** è un sottotipo più potente di **Iterator**, disponibile soltanto per le classi **List**. A differenza di **Iterator**, che può procedere soltanto in avanti, **ListIterator** è bidirezionale, e può anche fornire gli indici degli elementi successivi e precedenti rispetto al punto in cui si trova l'iteratore, nonché sostituire l'ultimo elemento “visitato” ricorrendo al metodo `set()`. Potete creare un **ListIterator** che punta all'inizio della **List**, chiamando `listIterator()`, oppure che inizia l'iterazione da un indice **n** nella lista, chiamando `listIterator(n)`. L'esempio seguente illustra queste possibilità.

```
//: holding/ListIteration.java
import typeinfo.pets.*;
import java.util.*;
```



```

public class ListIteration {
    public static void main(String[] args) {
        List<Pet> pets = Pets.arrayList(8);
        ListIterator<Pet> it = pets.listIterator();
        while(it.hasNext())
            System.out.print(it.next() + ", " + it.nextInt() +
                ", " + it.previousIndex() + "; ");
        System.out.println();
        // All'indietro:
        while(it.hasPrevious())
            System.out.print(it.previous().id() + " ");
        System.out.println();
        System.out.println(pets);
        it = pets.listIterator(3);
        while(it.hasNext()) {
            it.next();
            it.set(Pets.randomPet());
        }
        System.out.println(pets);
    }
} /* Output:
Rat, 1, 0; Manx, 2, 1; Cymric, 3, 2; Mutt, 4, 3; Pug, 5, 4;
Cymric, 6, 5; Pug, 7, 6; Manx, 8, 7;
7 6 5 4 3 2 1 0
[Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug, Manx]
[Rat, Manx, Cymric, Cymric, Rat, EgyptianMau, Hamster,
EgyptianMau]
*///:~

```

Il metodo **Pets.randomPet()** è impiegato per sostituire tutti gli oggetti **Pet** in **List**, a partire dalla posizione 3.

Esercizio 12 (3) Create e popolate una **List<Integer>**. Create una seconda **List<Integer>** con dimensioni identiche alla precedente e utilizzate **ListIterators** per leggere gli elementi dalla prima lista e inserirli nella seconda, in ordine inverso: tenete presente che sono possibili diverse soluzioni.

implementazione di
una classe
che rappresenta un
oggetto del tipo
“animale”.



LinkedList

Come **ArrayList**, anche **LinkedList** implementa l’interfaccia di base **List**, tuttavia esegue le operazioni di inserimento e rimozione all’interno dell’elenco in modo più efficiente di **ArrayList**; di contro è meno potente nelle operazioni ad accesso casuale.

LinkedList aggiunge anche metodi che permettono di utilizzarla come uno stack, una coda (**Queue**) o una “coda doppia” (*double-ended queue*, o **deque**).

Alcuni di questi metodi sono alias o minime variazioni l’uno dell’altro, allo scopo di produrre metodi dai nomi più familiari all’interno di un particolare contesto di utilizzo: **Queue**, in particolare.

Per esempio, **getFirst()** ed **element()** sono identici: entrambi restituiscono, senza rimuoverla, la “testa” (*head*, ossia il primo elemento della lista), sollevando un’eccezione di tipo **NoSuchElementException** nel caso la **List** sia vuota; **peek()** è una variante di questi due metodi, che restituisce **null** se la lista è vuota.

Anche **removeFirst()** e **remove()** sono identici: rimuovono e restituiscono il primo elemento della lista e sollevano anch’essi un’eccezione **NoSuchElementException** se l’elenco è vuoto, mentre **poll()** è una variante che restituisce **null** se la lista è vuota.

Il metodo **addFirst()** inserisce un elemento all’inizio dell’elenco.

Il metodo **offer()** è identico ad **add()** e **addLast()**: tutti e tre aggiungono un elemento alla “coda” (*tail*) della lista.

Il metodo **removeLast()** rimuove e restituisce l’ultimo elemento dell’elenco.

Di seguito è mostrato un esempio che evidenzia la somiglianza e le differenze di base tra queste funzionalità, senza ripetere il comportamento illustrato in **ListFeatures.java**.

```

//: holding/LinkedListFeatures.java
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class LinkedListFeatures {
    public static void main(String[] args) {
        LinkedList<Pet> pets =
            new LinkedList<Pet>(Pets.arrayList(5));
        print(pets);
    }
}

```



```

// Identici:
print("pets.getFirst(): " + pets.getFirst());
print("pets.element(): " + pets.element());
// Differisce soltanto in caso di lista vuota:
print("pets.peek(): " + pets.peek());
// Identici; rimuove e restituisce il primo elemento:
print("pets.remove(): " + pets.remove());
print("pets.removeFirst(): " + pets.removeFirst());
// Differisce soltanto in caso di lista vuota:
print("pets.poll(): " + pets.poll());
print(pets);
pets.addFirst(new Rat());
print("After addFirst(): " + pets);
pets.offer(Pets.randomPet());
print("After offer(): " + pets);
pets.add(Pets.randomPet());
print("After add(): " + pets);
pets.addLast(new Hamster());
print("After addLast(): " + pets);
print("pets.removeLast(): " + pets.removeLast());
}
} /* Output:
[Rat, Manx, Cymric, Mutt, Pug]
pets.getFirst(): Rat
pets.element(): Rat
pets.peek(): Rat
pets.remove(): Rat
pets.removeFirst(): Manx
pets.poll(): Cymric
[Mutt, Pug]
After addFirst(): [Rat, Mutt, Pug]
After offer(): [Rat, Mutt, Pug, Cymric]
After add(): [Rat, Mutt, Pug, Cymric, Pug]
After addLast(): [Rat, Mutt, Pug, Cymric, Pug, Hamster]
pets.removeLast(): Hamster
*///:~

```



Il risultato di **Pets.arrayList()** viene passato al costruttore di **LinkedList** per popolarlo.

Se esaminate l'interfaccia **Queue** noterete i metodi **element()**, **offer()**, **peek()**, **poll()** e **remove()** che sono stati aggiunti a **LinkedList** per trasformarla in un'implementazione di **Queue**. Nel prosieguo del capitolo troverete gli esempi completi di **Queue**.

Esercizio 13 (3) Nell'esempio **innerclasses/GreenhouseController.java** la classe **Controller** utilizza un **ArrayList**; modificate il codice per utilizzare una **LinkedList** e un **Iterator** per iterare l'insieme degli eventi.

Esercizio 14 (3) Generate una **LinkedList<Integer>** vuota e tramite un **ListIterator** aggiungete **Integer** alla **List**, inserendoli sempre nel mezzo della lista.

Stack

Lo stack, o pila, è definito come un contenitore di tipo LIFO (*Last-In, First-Out*), e talvolta chiamato *stack di tipo pushdown*, poiché qualunque cosa vi “introduciate” per ultima sarà la prima che potrete “estrarvi”. Un'analogia cui si ricorre spesso è quella delle pile di vassoi in una mensa: gli ultimi che vi vengono appoggiati sono i primi a essere estratti.

LinkedList dispone di metodi che implementano direttamente le funzionalità di stack, pertanto potreste anche utilizzare una **LinkedList** invece di realizzare una classe di stack. Tuttavia una classe di stack può rivelarsi talvolta più utile.

```

//: net/mindview/util/Stack.java
// Genera uno stack da una LinkedList.
package net.mindview.util;
import java.util.LinkedList;

public class Stack<T> {
    private LinkedList<T> storage = new LinkedList<T>();
    public void push(T v) { storage.addFirst(v); }
    public T peek() { return storage.getFirst(); }
    public T pop() { return storage.removeFirst(); }
    public boolean empty() { return storage.isEmpty(); }
    public String toString() { return storage.toString(); }
} //:~

```

Questo codice introduce l'esempio più semplice di una definizione di classe, utilizzando i generici: la <T> dopo il nome della classe indica al compilatore che si tratta di un tipo parametrato (*parameterized type*) e che il parametro indicante il tipo, vale a dire quello che sarà sostituito con il tipo effettivo al momento di utilizzare la classe, è T. Fondamentalmente, questo equivale a ordinare al compilatore di definire uno **Stack** contenente oggetti di tipo T. Lo **Stack** è implementato per mezzo di una **LinkedList**, alla quale viene chiesto di gestire il tipo T. Notate che il metodo **push()** accetta un oggetto di tipo T, mentre **peek()** e **pop()** restituiscono un oggetto di tipo T. Il metodo **peek()** fornisce l'elemento superiore, ossia il primo disponibile, senza rimuoverlo dall'inizio dello stack, mentre **pop()** rimuove e restituisce l'elemento superiore.

Per beneficiare unicamente del comportamento dello stack l'ereditarietà non è appropriata, poiché produrrebbe una classe con tutti gli altri metodi di **LinkedList**: nel Volume 2, Capitolo 5 vedrete che questo è esattamente l'errore commesso anche dai tecnici di Java 1.0 quando hanno progettato **java.util.Stack**.

Ecco una semplice dimostrazione di questa nuova classe **Stack**.

```
//: holding/StackTest.java
import net.mindview.util.*;

public class StackTest {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<String>();
        for(String s : "My dog has fleas".split(" "))
            stack.push(s);
        while(!stack.empty())
            System.out.print(stack.pop() + " ");
    }
} /* Output:
fleas has dog My
fleas has dog My
*///:~
```

Se volete utilizzare questa classe **Stack** nel vostro codice, dovete specificare il nome completo del package o cambiare il nome della classe al momento della creazione di quest'ultima; in caso contrario vi ritroverete a collidere con lo **Stack** del pacchetto **java.util**. Per esempio, se eseguite **import java.**

11 • Come contenere gli oggetti

util.* nell'esempio precedente, per evitare collisioni dovete utilizzare i nomi dei pacchetti.

```
//: holding/StackCollision.java
import net.mindview.util.*;

public class StackCollision {
    public static void main(String[] args) {
        net.mindview.util.Stack<String> stack =
            new net.mindview.util.Stack<String>();
        for(String s : "My dog has fleas".split(" "))
            stack.push(s);
        while(!stack.empty())
            System.out.print(stack.pop() + " ");
        System.out.println();
        java.util.Stack<String> stack2 =
            new java.util.Stack<String>();
        for(String s : "My dog has fleas".split(" "))
            stack2.push(s);
        while(!stack2.empty())
            System.out.print(stack2.pop() + " ");
    }
} /* Output:
fleas has dog My
fleas has dog My
*///:~
```

Le due classi **Stack** hanno la stessa interfaccia, ma non vi è alcuna interfaccia **Stack** comune in **java.util**, probabilmente perché la classe originale **java.util.Stack** di Java 1.0, mal progettata, ha assunto questo nome. Malgrado **java.util.Stack** esista, con **LinkedList** si ottiene uno **Stack** migliore, pertanto l'approccio **net.mindview.util.Stack** è preferibile.

Potete anche controllare la selezione dell'implementazione **Stack** "preferita" ricorrendo a un'importazione esplicita.

```
import net.mindview.util.Stack;
```



In questo modo qualsiasi riferimento a **Stack** selezionerà la versione di **net.mindview.util**, mentre per scegliere **java.util.Stack** dovete utilizzare il nome completo.

Esercizio 15 (4) Gli stack sono spesso utilizzati per valutare le espressioni nei linguaggi di programmazione. Servendovi di **net.mindview.util.Stack** valutate la seguente espressione, dove “+” vuol dire “spingere nello stack la lettera che segue”, e “-” significa “estrarre il primo elemento dello stack e visualizzarlo”: “+U+n+c---+e+r+t---+a-i-n+t+y---+r+u--+l+e+s---”.

Set

Un **Set** non ammette più di un’istanza dello stesso valore di un oggetto: se provate ad aggiungere più di una volta lo stesso oggetto il **Set** ve lo impedirà. L’utilizzo più comune di un **Set** è la “ricerca di un’appartenenza”, ovvero la verifica della presenza di un oggetto all’interno di un **Set**: per questo motivo la ricerca è l’operazione più importante che si esegue su un **Set**, che porta generalmente a preferire un’implementazione **HashSet**, ottimizzata per una ricerca rapida.

Set offre la stessa interfaccia di **Collection**, quindi non presenta alcuna funzionalità aggiuntiva come accade in due tipi diversi di **List**. **Set** è esattamente una **Collection**, ma con un diverso comportamento; come vedete, questa è l’applicazione ideale dell’ereditarietà e del polimorfismo: esprimere comportamenti differenti. Un **Set** determina l’appartenenza basandosi sul “valore” di un oggetto, un argomento piuttosto complesso che avrete modo di approfondire nel Volume 2, Capitolo 5.

Osservate l’esempio seguente che utilizza un **HashSet** di oggetti **Integer**.

```
//: holding/SetOfInteger.java
import java.util.*;

public class SetOfInteger {
    public static void main(String[] args) {
        Random rand = new Random(47);
        Set<Integer> intset = new HashSet<Integer>();
        for(int i = 0; i < 10000; i++)
            intset.add(rand.nextInt(30));
        System.out.println(intset);
    }
}
```

```
} /* Output:
[15, 8, 23, 16, 7, 22, 9, 21, 6, 1, 29, 14, 24, 4, 19, 26, 11,
18, 3, 12, 27, 17, 2, 13, 28, 20, 25, 10, 5, 0]
*///:~
```

Al **Set** vengono aggiunti 10.000 numeri casuali nell’intervallo da 0 a 29, in modo che per ogni valore esistano sicuramente molte occorrenze: come potete vedere, però, nel risultato appare soltanto un’istanza di ciascun numero.

Avrete anche osservato che l’output non presenta alcun ordine apparente. Questo accade perché, per motivi di prestazioni, **HashSet** adotta un algoritmo di *hashing* a indirizzamento casuale: un altro argomento che vedrete meglio nel Volume 2, Capitolo 5. L’ordine mantenuto da un **HashSet** è differente da quello di un **TreeSet** o di un **LinkedHashSet**, poiché ogni implementazione immagazzina gli elementi in modo diverso. **TreeSet** mantiene gli elementi ordinati in una struttura di dati ad albero, in cui ciascun nodo è associato ai colori rosso o nero, secondo regole appropriate, mentre **HashSet** impiega una funzione hashing a indirizzamento casuale. Anche **LinkedHashSet** si serve dell’algoritmo di hashing per velocizzare la ricerca, tuttavia sembra che mantenga gli elementi in ordine d’inserimento tramite una lista collegata.

Per fare in modo che i risultati appaiano in ordine, una tecnica possibile è l’impiego di un **TreeSet** in luogo di un **HashSet**.

```
//: holding/SortedSetOfInteger.java
import java.util.*;

public class SortedSetOfInteger {
    public static void main(String[] args) {
        Random rand = new Random(47);
        SortedSet<Integer> intset = new TreeSet<Integer>();
        for(int i = 0; i < 10000; i++)
            intset.add(rand.nextInt(30));
        System.out.println(intset);
    }
} /* 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]
*///:~
```



Una delle operazioni che eseguirete con maggiore frequenza è il test di appartenenza a un insieme utilizzando il metodo **contains()**; esistono tuttavia anche operazioni che ricorderanno i diagrammi di Venn a chi ha una competenza matematica.

```
//: holding/SetOperations.java
import java.util.*;
import static net.mindview.util.Print.*;

public class SetOperations {
    public static void main(String[] args) {
        Set<String> set1 = new HashSet<String>();
        Collections.addAll(set1,
            "A B C D E F G H I J K L".split(" "));
        set1.add("M");
        print("H: " + set1.contains("H"));
        print("N: " + set1.contains("N"));
        Set<String> set2 = new HashSet<String>();
        Collections.addAll(set2, "H I J K L".split(" "));
        print("set2 in set1: " + set1.containsAll(set2));
        set1.remove("H");
        print("set1: " + set1);
        print("set2 in set1: " + set1.containsAll(set2));
        set1.removeAll(set2);
        print("set2 removed from set1: " + set1);
        Collections.addAll(set1, "X Y Z".split(" "));
        print("'X Y Z' added to set1: " + set1);
    }
} /* Output:
H: true
N: false
set2 in set1: true
set1: [D, K, C, B, L, G, I, M, A, F, J, E]
set2 in set1: false
set2 removed from set1: [D, C, B, G, M, A, F, E]
'X Y Z' added to set1: [Z, D, C, B, G, M, A, F, Y, X, E]
*///:~
```



I nomi dei metodi non richiedono spiegazioni: ne troverete altri nella documentazione JDK.

Elaborare un elenco di elementi univoci può essere abbastanza utile.

Supponete, per esempio, di volere elencare tutte le parole nel file **SetOperations.java**, cui si riferisce il codice precedente; utilizzando il programma di utilità **net.mindview.TextFile**, che sarà introdotto nel prosieguo del manuale, potete aprire e leggere un file in un **Set**.

```
//: holding/UniqueWords.java
import java.util.*;
import net.mindview.util.*;

public class UniqueWords {
    public static void main(String[] args) {
        Set<String> words = new TreeSet<String>(
            new TextFile("SetOperations.java", "\\\\W+"));
        System.out.println(words);
    }
} /* Output:
[A, B, C, Collections, D, E, F, G, H, HashSet, I, J, K, L,
M, N, Output, Print, Set, SetOperations, String, X, Y, Z, to,
add, addAll, added, args, class, contains, containsAll, from,
removed, false, holding, import, in, java, main, mindview,
net, new, print, public, remove, removeAll, set1, set2, split,
static, true, util, void]
*///:~
```

TextFile è ereditato da **List**. Il costruttore di **TextFile** apre il file e lo seziona in parole (sequenze di lettere), in conformità all'espressione regolare "**\W+**" che significa "una o più lettere": le espressioni regolari verranno introdotte nel Capitolo 12, dedicato alle stringhe.

Il risultato viene passato al costruttore di **TreeSet**, che aggiunge a se stesso il contenuto di **List**.

Trattandosi di un **TreeSet** il risultato viene ordinato: in questo caso l'ordinamento è di tipo *lessicografico*, nel quale le lettere maiuscole e minuscole risultano in gruppi separati.



Se preferite che l'elenco sia ordinato alfabeticamente potete passare al costruttore di **TreeSet** il **Comparator String.CASE_INSENSITIVE_ORDER**.

```
//: holding/UniqueWordsAlphabetic.java
// Produce un elenco alfabetico.
import java.util.*;
import net.mindview.util.*;

public class UniqueWordsAlphabetic {
    public static void main(String[] args) {
        Set<String> words =
            new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
        words.addAll(
            new TextFile("SetOperations.java", "\\\W+"));
        System.out.println(words);
    }
} /* Output:
[A, add, addAll, aggiunto, args, B, C, class, Collections,
contains, containsAll, D, da, E, eliminato, F, false, G, H,
HashSet, holding, I, import, in, J, java, K, L, M, main,
mindview, N, net, new, Output, Print, public, remove,
removeAll, Set, set1, set2, SetOperations, split, static,
String, true, util, void, X, Y, Z]
*///:~
```

Un **Comparator** è un oggetto che stabilisce un tipo di ordinamento, che potrete vedere in dettaglio nel Volume 2, Capitolo 4.

Esercizio 16 (5) Create un **Set** contenente le vocali. Partendo da **UniqueWords.java** contate e visualizzate il numero di vocali in ogni parola di input, visualizzando anche il numero totale di vocali presenti nel file di input.

Map

La capacità di eseguire una corrispondenza tra gli oggetti, il cosiddetto *mapping*, può rivelarsi una tecnica di enorme importanza per risolvere alcuni problemi di programmazione. Considerate, per esempio, un programma che analizza la casualità della classe **Random** di Java: sostanzialmen-

te **Random** produce una distribuzione perfetta dei numeri, tuttavia per dimostrare tale “perfezione” è necessario generare molti numeri casuali e contare quelli che rientrano nei diversi intervalli. Una **Map** permette di risolvere facilmente il problema; in questo caso, la chiave è il numero prodotto da **Random** e il valore è il numero di volte che il numero viene visualizzato.

```
//: holding/Statistics.java
// Semplice dimostrazione di HashMap.
import java.util.*;

public class Statistics {
    public static void main(String[] args) {
        Random rand = new Random(47);
        Map<Integer, Integer> m =
            new HashMap<Integer, Integer>();
        for(int i = 0; i < 10000; i++) {
            // Produce un numero tra 0 e 20:
            int r = rand.nextInt(20);
            Integer freq = m.get(r);
            m.put(r, freq == null ? 1 : freq + 1);
        }
        System.out.println(m);
    }
} /* Output:
{15=497, 4=481, 19=464, 8=468, 11=531, 16=533, 18=478, 3=508,
7=471, 12=521, 17=509, 2=489, 13=506, 9=549, 6=519, 1=502,
14=477, 10=513, 5=503, 0=481}
*///:~
```

In **main()** la funzionalità di autoboxing converte l'**int** generato casualmente in un riferimento **Integer** utilizzabile con **HashMap**: ricordate che con i contenitori non è permesso utilizzare i tipi primitivi. Il metodo **get()** restituisce **null** se la chiave non è già presente nel contenitore, per indicare che il numero è stato trovato per la prima volta; diversamente il metodo **get()** produrrà il valore **Integer** associato alla chiave, che verrà incrementato (anche in questo caso l'autoboxing semplifica l'espressione, tuttavia si verificano comunque conversioni verso e da **Integer**).



Di seguito è presentato un esempio che consente di utilizzare una descrizione **String** per cercare gli oggetti **Pet**, e mostra anche come sia possibile testare una **Map** per verificare se contiene una chiave o un valore, mediante **containsKey()** e **containsValue()**.

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

public class PetMap {
    public static void main(String[] args) {
        Map<String,Pet> petMap = new HashMap<String,Pet>();
        petMap.put("My Cat", new Cat("Molly"));
        petMap.put("My Dog", new Dog("Ginger"));
        petMap.put("My Hamster", new Hamster("Bosco"));
        print(petMap);
        Pet dog = petMap.get("My Dog");
        print(dog);
        print(petMap.containsKey("My Dog"));
        print(petMap.containsValue(dog));
    }
} /* Output:
{ My Cat=Cat Molly, My Hamster=Hamster Bosco, My Dog=Dog
Ginger}
Dog Ginger
true
true
*//*:~
```

Proprio come è possibile fare con gli array e le **Collection**, le dimensioni delle **Map** possono essere facilmente espanso: a questo scopo è sufficiente creare una mappa i cui valori siano altre **Map** e i valori di queste ultime altri contenitori, anche altre mappe.

In questo modo non è difficile combinare contenitori che producano velocemente strutture dati potenti. Per esempio, immaginate di dovere tenere trac-



cia delle persone che possiedono diversi animali domestici (*pet*); in tal caso vi basterà disporre di una **Map<Person, List<Pet>>**:

```
//: holding/MapOfList.java
package holding;
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class MapOfList {
    public static Map<Person, List<? extends Pet>>
        petPeople = new HashMap<Person, List<? extends Pet>>();
    static {
        petPeople.put(new Person("Dawn"),
            Arrays.asList(new Cymric("Molly"),new Mutt("Spot")));
        petPeople.put(new Person("Kate"),
            Arrays.asList(new Cat("Shackleton"),
                new Cat("Elsie May"), new Dog("Margrett")));
        petPeople.put(new Person("Marilyn"),
            Arrays.asList(
                new Pug("Louie aka Louis Snorkelstein Dupree"),
                new Cat("Stanford aka Stinky el Negro"),
                new Cat("Pinkola")));
        petPeople.put(new Person("Luke"),
            Arrays.asList(new Rat("Fuzzy"), new Rat("Fizzy")));
        petPeople.put(new Person("Isaac"),
            Arrays.asList(new Rat("Freckly")));
    }
    public static void main(String[] args) {
        print("People: " + petPeople.keySet());
        print("Pets: " + petPeople.values());
        for(Person person : petPeople.keySet()) {
            print(person + " has:");
            for(Pet pet : petPeople.get(person))
                print(" " + pet);
        }
    }
}
```

```

} /* Output:
People: [Person Luke, Person Marilyn, Person Isaac, Person
Dawn, Person Kate]
Pets: [[Rat Fuzzy, Rat Fizzy], [Pug Louie aka Louis
Snorkelstein Dupree, Cat Stanford aka Stinky el Negro, Cat
Pinkola], [Rat Freckly], [Cymric Molly, Mutt Spot], [Cat
Shackleton, Cat Elsie May, Dog Margrett]]
Person Luke has:
    Rat Fuzzy
    Rat Fizzy
Person Marilyn has:
    Pug Louie aka Louis Snorkelstein Dupree
    Cat Stanford aka Stinky el Negro
    Cat Pinkola
Person Isaac has:
    Rat Freckly
Person Dawn has:
    Cymric Molly
    Mutt Spot
Person Kate has:
    Cat Shackleton
    Cat Elsie May
    Dog Margrett
*///:~

```

Una **Map** può restituire un **Set** delle sue chiavi, una **Collection** dei suoi valori o un **Set** delle sue coppie di chiave/valore.

Il metodo **keySet()** produce un **Set** di tutte le chiavi in **petPeople**, utilizzato nell'istruzione **foreach** per iterare la mappa.

Esercizio 17 (2) Prendete la classe **Gerbil** dell'Esercizio 1 e inseritela in una **Map**, associando il nome di ogni **Gerbil** (per esempio "Spot" o "Fuzzy") a una **String** (la chiave) per ogni gerbillo inserito (il valore). Quindi ottenete un **Iterator** per il **keySet()** e utilizzatelo per spostarvi nella **Map**: cercate il **Gerbil** corrispondente a ogni chiave, visualizzate-la e ordinate al gerbillo di "saltare" (**hop()**).

Esercizio 18 (3) Popolate una **HashMap** di coppie chiave/valore e vi-visualizzate i risultati, per evidenziare l'ordinamento secondo il codice hash. Estraete le coppie, ordinatele in base alla chiave e collocate i

risultati in una **LinkedHashMap**, visualizzando che l'ordine d'inserimen-to è stato conservato.

Esercizio 19 (2) Ripetete l'esercizio precedente con un **HashSet** e un **LinkedHashSet**.

Esercizio 20 (3) Modificate l'Esercizio 16 in modo da mantenere un conteggio delle occorrenze di ogni vocale.

Esercizio 21 (3) Utilizzando una **Map<String, Integer>** seguite il mo-dello di **UniqueWords.java** per creare un programma che conta l'oc-correnza delle parole contenute in un file. Ordinate i risultati mediante **Collections.sort()** con un secondo argomento **String.CASE_INSEN-SITIVE_ORDER** (per produrre un ordinamento alfabetico) e visua-lizzate il risultato.

Esercizio 22 (5) Modificate l'esercizio precedente in modo che utilizzi una classe contenente una **String** e un campo di conteggio per me-morizzare ogni parola diversa, e un **Set** di questi oggetti per gestire l'elenco di parole.

Esercizio 23 (4) Partendo da **Statistics.java** create un programma che esegue il test ripetutamente e controlla se nei risultati un numero ricorre più spesso di un altro.

Esercizio 24 (2) Popolate una **LinkedHashMap** di chiavi **String** e og-getti di vostra scelta, poi estraete le coppie, ordinatele in base alle chia-vi e inseritele nella **Map**.

Esercizio 25 (3) Create una **Map<String, ArrayList<Integer>>**. Utilizza-te **net.mindview.TextFile** per aprire un file di testo e leggetelo una parola alla volta: servitevi di "**\W+**" come secondo argomento del costruttore di **TextFile**. Contate le parole via via che le leggete, e per ogni parola nel file registrate in **ArrayList<Integer>** il conteggio relativo: questa è, in effetti, la posizione del file in cui si trova la parola stessa.

Esercizio 26 (4) Prendendo la **Map** risultante dall'esercizio precedente, ricreate l'ordine delle parole così come apparivano nel file originale.

Queue

Una coda o *queue* è tipicamente un contenitore di tipo FIFO (*First-In, First-Out*), nel quale si introducono elementi in un'estremità che vengono poi estratti dall'altra nello stesso ordine in cui sono stati introdotti. Di norma le code sono utilizzate per trasferire in modo attendibile gli oggetti da un'area di un programma a un'altra, e sono un componente importante della cosid-

della programmazione concorrente (Volume 3, Capitolo 1) poiché consentono di trasferire in assoluta sicurezza gli oggetti da un task all'altro.

LinkedList dispone di metodi che supportano il comportamento delle code e implementa l'interfaccia **Queue**: può quindi essere utilizzata come un'implementazione della coda. Eseguendo l'upcasting di una **LinkedList** a **Queue**, l'esempio seguente utilizza i metodi specifici per le code nell'interfaccia.

```
//: holding/QueueDemo.java
// Upcasting da una LinkedList a una Queue.
import java.util.*;

public class QueueDemo {
    public static void printQ(Queue queue) {
        while(queue.peek() != null)
            System.out.print(queue.remove() + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<Integer>();
        Random rand = new Random(47);
        for(int i = 0; i < 10; i++)
            queue.offer(rand.nextInt(i + 10));
        printQ(queue);
        Queue<Character> qc = new LinkedList<Character>();
        for(char c : "Brontosaurus".toCharArray())
            qc.offer(c);
        printQ(qc);
    }
} /* Output:
8 1 1 1 5 14 3 1 0 1
Brontosaurus
*///:~
```

Il metodo **offer()** è uno di quelli specifici per le code: inserisce un elemento in coda alla **Queue**, restituendo **false** qualora tale operazione non sia possibile. Sia **peek()** sia **element()** restituiscono la testa della coda senza rimuoverla, tuttavia **peek()** restituisce **null** se la coda è vuota, mentre **element()** solleva

un'eccezione di tipo **NoSuchElementException**; i metodi **poll()** e **remove()** rimuovono e restituiscono la testa della coda, ma **poll()** restituisce **null** se la coda è vuota, mentre **remove()** solleva un'eccezione **NoSuchElementException**.

La funzionalità di autoboxing si incarica di convertire automaticamente il risultato **int** di **nextInt()** nell'oggetto **Integer** richiesto da **queue**, e il **char** **c** nell'oggetto **Character** richiesto da **qc**. L'interfaccia **Queue** limita l'accesso ai metodi di **LinkedList** in modo che soltanto i metodi appropriati siano disponibili: così facendo sarete meno invogliati a utilizzare i metodi di **LinkedList**. In realtà potete sempre sottoporre a cast la coda per trasformarla di nuovo in una **LinkedList**, tuttavia sarebbe preferibile evitare questa operazione.

Notate che i metodi specifici di **Queue** forniscono funzionalità complete e indipendenti: potete infatti avere una **Queue** utilizzabile senza nessuno dei metodi disponibili in **Collection**, da cui eredita la coda.

Esercizio 27 (2) Scrivete una classe chiamata **Command** che contiene una **String** e un metodo **operation()** che visualizza la stringa. Scrivete una seconda classe con un metodo che popola una **Queue** di oggetti **Command** e la restituisce. Infine passate la **Queue** a un metodo di una terza classe che “consuma” gli oggetti in **Queue** e chiama i loro metodi **operation()**.

PriorityQueue

Una disciplina di accodamento (*queuing discipline*) determina quale degli elementi presenti in una coda sarà considerato come elemento successivo. L'espressione “*first-in, first-out*” (*FIFO*) indica che l'elemento successivo dovrebbe essere quello che è rimasto in attesa più a lungo.

Una coda con priorità (*priority queue*) segnala invece che il successivo elemento sarà quello con la priorità più elevata: per esempio, in un aeroporto un cliente potrebbe essere “prelevato” da una coda se il suo aereo è in partenza. In un sistema di messaggistica, alcuni messaggi saranno più importanti e dovranno essere letti prima degli altri, indipendentemente dall'ordine di arrivo. La **PriorityQueue** è stata aggiunta in Java SE5 per fornire un'implementazione automatica di questo comportamento.

Quando “porgete” (**offer()**) un oggetto a una **PriorityQueue** esso viene messo in ordine all'interno della coda.⁵

5. In realtà questo dipende dall'implementazione: negli algoritmi delle code di priorità l'ordinamento avviene di solito al momento dell'inserimento, tramite un heap, tuttavia alcuni algoritmi potrebbero selezionare gli elementi prioritari anche durante la rimozione. La scelta dell'algoritmo può essere decisiva qualora la priorità dell'oggetto cambi mentre questo è in attesa.

Il criterio di ordinamento predefinito si basa sull'ordine naturale degli oggetti in coda, che potete comunque modificare fornendo il vostro **Comparator**. La **PriorityQueue** vi garantisce che al momento di chiamare **peek()**, **poll()** o **remove()** l'elemento ottenuto sarà quello a priorità più elevata.

Non è difficile realizzare una **PriorityQueue** che funziona con i tipi nativi, quali **Integer**, **String** o **Character**; nell'esempio che segue il primo insieme di valori corrisponde esattamente ai valori casuali utilizzati nell'esempio precedente, per sottolineare il loro diverso trattamento in una **PriorityQueue**.

```
//: holding/PriorityQueueDemo.java
import java.util.*;

public class PriorityQueueDemo {
    public static void main(String[] args) {
        PriorityQueue<Integer> priorityQueue =
            new PriorityQueue<Integer>();
        Random rand = new Random(47);
        for(int i = 0; i < 10; i++)
            priorityQueue.offer(rand.nextInt(i + 10));
        QueueDemo.printQ(priorityQueue);

        List<Integer> ints = Arrays.asList(25, 22, 20,
            18, 14, 9, 3, 1, 1, 2, 3, 9, 14, 18, 21, 23, 25);
        priorityQueue = new PriorityQueue<Integer>(ints);
        QueueDemo.printQ(priorityQueue);
        priorityQueue = new PriorityQueue<Integer>(
            ints.size(), Collections.reverseOrder());
        priorityQueue.addAll(ints);
        QueueDemo.printQ(priorityQueue);

        String fact = "EDUCATION SHOULD ESCHEW OBFUSCATION";
        List<String> strings = Arrays.asList(fact.split(""));
        PriorityQueue<String> stringPQ =
            new PriorityQueue<String>(strings);
        QueueDemo.printQ(stringPQ);
    }
}
```

```
stringPQ = new PriorityQueue<String>(
    strings.size(), Collections.reverseOrder());
stringPQ.addAll(strings);
QueueDemo.printQ(stringPQ);

Set<Character> charSet = new HashSet<Character>();
for(char c : fact.toCharArray())
    charSet.add(c); // Autoboxing
PriorityQueue<Character> characterPQ =
    new PriorityQueue<Character>(charSet);
QueueDemo.printQ(characterPQ);
}

} /* Output:
0 1 1 1 1 3 5 8 14
1 1 2 3 3 9 9 14 14 18 18 20 21 22 23 25 25
25 25 23 22 21 20 18 18 14 14 9 9 3 3 2 1 1
A A B C C C D D E E E F F H H I I L N N O O O O S S S
T T U U U W
W U U U T T S S S O O O O N N L I I H H F E E E D D C C C B
A A
A B C D E F H I L N O S T U W
*///:~
```

Potete vedere che i duplicati sono permessi e che i valori più bassi hanno la priorità più elevata; nel caso di **String** anche gli spazi contano come valori, e la loro priorità è superiore rispetto alle lettere. Per illustrarvi la possibilità di modificare l'ordine fornendo un oggetto **Comparator** personalizzato, la terza chiamata al costruttore a **PriorityQueue<Integer>** e la seconda chiamata a **PriorityQueue<String>** utilizzano entrambe il **Comparator** in ordine inverso prodotto da **Collections.reverseOrder()**, disponibile in Java SE5.

L'ultima sezione del codice aggiunge un **HashSet** per eliminare i **Character** duplicati, in modo da rendere l'esempio più interessante.

Le classi **Integer**, **String** e **Character** funzionano con **PriorityQueue** poiché sono già predisposte per l'ordinamento naturale. Se volete utilizzare la vostra classe in una **PriorityQueue** dovete includere funzionalità aggiuntive che producano l'ordine naturale, oppure fornire il vostro **Comparator**.

Nel Volume 2, Capitolo 5 sarà presentato un esempio più complesso che dimostra quanto affermato.

Esercizio 28 (2) Ricorrendo a `offer()` popolate una **PriorityQueue** con valori **Double** creati mediante **java.util.Random**, quindi rimuovete gli elementi utilizzando `poll()` e visualizzateli.

Esercizio 29 (2) Create una semplice classe che eredita da **Object**, priva di membri, e dimostrate che è impossibile aggiungere elementi multipli di quella classe a una **PriorityQueue**: questo argomento sarà analizzato in dettaglio nel Volume 2, Capitolo 5.

Confronto tra Collection e Iterator

Collection è l’interfaccia radice che descrive le funzionalità comuni a tutti i contenitori di sequenze: potrebbe essere considerata come un “interfaccia casuale”, che è disponibile per analogia con altre interfacce. La classe **java.util.AbstractCollection** fornisce anche un’implementazione predefinita per una **Collection**, che consente di creare un nuovo sottotipo di **AbstractCollection** senza inutile duplicazione di codice.

Uno dei motivi che giustificano l’utilità di un’interfaccia è la possibilità di generare codice più generico: scrivendo un’interfaccia invece che un’implementazione il vostro codice può essere applicato a più tipi di oggetti.⁶

Quindi, se scrivete un metodo che accetta una **Collection**, esso potrà essere applicato a qualunque tipo implementi **Collection**: questa caratteristica fa sì che una nuova classe possa scegliere di implementare **Collection** al fine di essere utilizzata con il vostro metodo. È interessante notare, tuttavia, che la libreria standard di C++ non dispone di una classe di base comune per i propri contenitori: tutte le analogie tra i contenitori sono gestite per mezzo di iteratori. In Java potrebbe sembrare ragionevole adottare la tecnica utilizzata in C++ per esprimere le proprietà comuni ai contenitori, ricorrendo a un iteratore anziché a una **Collection**; in ogni caso i due metodi sono collegati tra loro, di conseguenza implementare una **Collection** significa anche fornire un metodo **iterator()**.

```
//: holding/InterfaceVsIterator.java
```

6. Alcuni programmati sostengono la necessità di creare automaticamente interfacce per qualsiasi combinazione di metodi in una classe, talvolta per ogni classe. È opinione dell’autore che un’interfaccia debba essere qualcosa di più di una duplicazione meccanica di combinazioni di metodi e che, pertanto, sia opportuno verificare il potenziale valore aggiunto di un’interfaccia prima di implementarla.

```
import typeinfo.pets.*;
import java.util.*;

public class InterfaceVsIterator {
    public static void display(Iterator<Pet> it) {
        while(it.hasNext()) {
            Pet p = it.next();
            System.out.print(p.id() + ":" + p + " ");
        }
        System.out.println();
    }

    public static void display(Collection<Pet> pets) {
        for(Pet p : pets)
            System.out.print(p.id() + ":" + p + " ");
        System.out.println();
    }

    public static void main(String[] args) {
        List<Pet> petList = Pets.arrayList(8);
        Set<Pet> petSet = new HashSet<Pet>(petList);
        Map<String,Pet> petMap =
            new LinkedHashMap<String,Pet>();
        String[] names = ("Ralph, Eric, Robin, Lacey, " +
            "Britney, Sam, Spot, Fluffy").split(", ");
        for(int i = 0; i < names.length; i++)
            petMap.put(names[i], petList.get(i));
        display(petList);
        display(petSet);
        display(petList.iterator());
        display(petSet.iterator());
        System.out.println(petMap);
        System.out.println(petMap.keySet());
        display(petMap.values());
        display(petMap.values().iterator());
    }
} /* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
```



```

4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx 2:Cymric 0:Rat
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx 2:Cymric 0:Rat
{Ralph=Rat, Eric=Manx, Robin=Cymric, Lacey=Mutt, Britney=Pug,
Sam=Cymric, Spot=Pug, Fluffy=Manx}
[Ralph, Eric, Robin, Lacey, Britney, Sam, Spot, Fluffy]
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
*///:~

```

Entrambe le versioni di **display()** funzionano con gli oggetti **Map** e i sottotipi di **Collection**, e sia l'interfaccia **Collection** sia l'**Iterator** fanno sì che i metodi **display()** non debbano disporre di informazioni sull'implementazione specifica del contenitore sottostante.

In questo caso i due approcci possono sembrare equivalenti: in realtà **Collection** potrebbe essere preferibile in quanto è **Iterable**, pertanto nell'esecuzione di **display(Collection)** è possibile utilizzare il ciclo `foreach`, che rende il codice leggermente più lineare.

L'utilizzo di **Iterator** diventa indispensabile quando occorra implementare una classe proveniente dall'esterno (*foreign class*), vale a dire non presente in una **Collection**, nella quale sarebbe complesso o seccante implementare l'interfaccia **Collection**.

Per esempio, se create un'implementazione di **Collection** ereditando da una classe che contiene oggetti **Pet** dovrete implementare tutti i metodi di **Collection**, anche se non tutti vi occorreranno nell'ambito del metodo **display()**; sebbene possiate portare a termine facilmente questa operazione ereditando da **AbstractCollection**, dovete comunque implementare **iterator()** e **size()** per fornire i metodi che non sono implementati da **AbstractCollection**, ma che sono utilizzati dagli altri metodi di **AbstractCollection**.

```

//: holding/CollectionSequence.java
import typeinfo.pets.*;
import java.util.*;

public class CollectionSequence
extends AbstractCollection<Pet> {
private Pet[] pets = Pets.createArray(8);
public int size() { return pets.length; }
public Iterator<Pet> iterator() {

```

```

return new Iterator<Pet>() {
private int index = 0;
public boolean hasNext() {
return index < pets.length;
}
public Pet next() { return pets[index++]; }
public void remove() { // Non implementato
throw new UnsupportedOperationException();
}
};

public static void main(String[] args) {
CollectionSequence c = new CollectionSequence();
InterfaceVsIterator.display(c);
InterfaceVsIterator.display(c.iterator());
}

/* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
*///:~

```

Il metodo **remove()** è una “operazione facoltativa” che vedrete meglio nel Volume 2, Capitolo 5: in questo caso specifico non è necessario implementarlo, e se lo chiamerete produrrà un'eccezione.

Dall'esempio risulta evidente che se si decide di implementare **Collection** occorre farlo anche per **iterator()**, e che la sola implementazione di questo metodo è poco meno complessa della semplice ereditarietà da **AbstractCollection**.

Tuttavia, se la vostra classe eredita da un'altra non potrete ereditare anche da **AbstractCollection**; in tal caso, per implementare **Collection** dovete implementare tutti i metodi dell'interfaccia.

In questo esempio specifico sarebbe molto più facile ricorrere all'ereditarietà e integrare la capacità di generare un iterator.

```

//: holding/NonCollectionSequence.java
import typeinfo.pets.*;
import java.util.*;


```



```

class PetSequence {
    protected Pet[] pets = Pets.createArray(8);
}

public class NonCollectionSequence extends PetSequence {
    public Iterator<Pet> iterator() {
        return new Iterator<Pet>() {
            private int index = 0;
            public boolean hasNext() {
                return index < pets.length;
            }
            public Pet next() { return pets[index++]; }
            public void remove() { // Non implementato
                throw new UnsupportedOperationException();
            }
        };
    }
    public static void main(String[] args) {
        NonCollectionSequence nc = new NonCollectionSequence();
        InterfaceVsIterator.display(nc.iterator());
    }
} /* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
*///:~

```

La creazione di un **Iterator** è la tecnica meno vincolante per realizzare il collegamento tra una sequenza e un metodo che la utilizza, e pone decisamente meno limiti alla classe di sequenza rispetto all'implementazione di una **Collection**.

Esercizio 30 (5) Modificate **CollectionSequence.java** in modo che non erediti da **AbstractCollection** ma implementi **Collection**.

Foreach e iteratori

Malgrado la sintassi foreach sia stata utilizzata finora soprattutto con gli array, funziona anche con qualsiasi oggetto **Collection**. Avete già visto questa

tecnicà all'opera utilizzando **ArrayList**, in ogni caso di seguito è mostrato un esempio generico.

```

//: holding/ForEachCollections.java
// Tutte le Collection funzionano con foreach.
import java.util.*;

public class ForEachCollections {
    public static void main(String[] args) {
        Collection<String> cs = new LinkedList<String>();
        Collections.addAll(cs,
            "Take the long way home".split(" "));
        for(String s : cs)
            System.out.print("'" + s + "' ");
    }
} /* Output:
'Take' 'the' 'long' 'way' 'home'
*///:~

```

Poiché **cs** è una **Collection** questo codice dimostra che tutti gli oggetti **Collection** funzionano con foreach.

Il motivo di un funzionamento così generalizzato è che Java SE5 ha introdotto una nuova interfaccia **Iterable** contenente un metodo **iterator()** che produce un **Iterator**, e che tale interfaccia è utilizzata da foreach per spostarsi all'interno di una sequenza. Di conseguenza, se create una classe che implementa **Iterable** potrete utilizzarla in un'istruzione foreach.

```

//: holding/IterableClass.java
// Fforeach funziona con qualsiasi oggetto sia Iterable.
import java.util.*;

public class IterableClass implements Iterable<String> {
    protected String[] words = ("And that is how " +
        "we know the Earth to be banana-shaped.").split(" ");
    public Iterator<String> iterator() {
        return new Iterator<String>() {
            private int index = 0;

```



```

public boolean hasNext() {
    return index < words.length;
}
public String next() { return words[index++]; }
public void remove() { // Non implementato
    throw new UnsupportedOperationException();
}
}
public static void main(String[] args) {
    for(String s : new IterableClass())
        System.out.print(s + " ");
}
} /* Output:
And that is how we know the Earth to be banana-shaped.
*///:~

```

Il metodo **iterator()** produce un'istanza di un'implementazione interna anonima di **Iterator<String>**, che restituisce ogni parola presente nell'array. In **main()** potete vedere che **IterableClass** opera effettivamente in un'istruzione **foreach**.

In Java SE5 molte classi sono state rese **Iterable**, principalmente classi **Collection** (ma non **Map**). Per esempio, il codice seguente visualizza tutte le variabili di ambiente del sistema operativo.

```

//: holding/EnvironmentVariables.java
import java.util.*;

public class EnvironmentVariables {
    public static void main(String[] args) {
        for(Map.Entry entry: System.getenv().entrySet()) {
            System.out.println(entry.getKey() + ":" +
                entry.getValue());
        }
    }
} /* (Da eseguire per visualizzare l'output) *///:~

```



System.getenv()⁷ restituisce una **Map**, **entrySet()** produce un **Set** di elementi **Map.Entry**, e **Set** è **Iterable** per essere utilizzabile in un ciclo **foreach**.

Un'istruzione **foreach** funziona con un array o qualsiasi componente **Iterable**, ma questo non significa che un array sia automaticamente **Iterable** né che lo sia l'esecuzione di **autoboxing**.

```

//: holding/ArrayIsNotIterable.java
import java.util.*;

public class ArrayIsNotIterable {
    static <T> void test(Iterable<T> ib) {
        for(T t : ib)
            System.out.print(t + " ");
    }
    public static void main(String[] args) {
        test(Arrays.asList(1, 2, 3));
        String[] strings = { "A", "B", "C" };
        // Un array funziona in foreach, tuttavia non e' Iterable:
        //! test(strings);
        // Deve essere convertito esplicitamente in un Iterable:
        test(Arrays.asList(strings));
    }
} /* Output:
1 2 3 A B C
*///:~

```

Se provate a passare un array come argomento **Iterable** vi renderete conto che non è possibile farlo. Java non esegue la conversione automatica a **Iterable**: dovrete occuparvene voi stessi.

Esercizio 31 (3) Modificate il codice di **polymorphism/shape/RandomShapeGenerator.java** per renderlo **Iterable**. Dovrete aggiungere un costruttore che accetta il numero di elementi che volete fare pro-

7. **System.getenv()** è un metodo disponibile soltanto a partire da Java SE5, perché in precedenza i progettisti Java lo ritenevano eccessivamente vincolato al sistema operativo, quindi in contrasto con il principio “scrivi una volta, esegui ovunque”. Il fatto che ora sia disponibile fa ritenere che il gruppo di sviluppo Java stia assumendo un'attitudine più pragmatica.



durre all'iteratore prima che questo si interrompa: verificatene il corretto funzionamento.

L'Adapter Method idiom

E se voleste arricchire una classe esistente di tipo **Iterable** di uno o più modi per utilizzarla in un'istruzione `foreach`? Supponete, per esempio, di volere scegliere se iterare un elenco di parole in avanti o all'indietro: limitandovi a ereditare dalla classe e sovrascrivere il metodo `iterator()` non otterrete altro che la sostituzione del metodo esistente.

Una soluzione possibile è ciò che l'autore ha definito *Adapter Method idiom*, letteralmente “idioma del metodo Adapter”. Il termine “Adapter” dell'espressione proviene dalle strutture *design pattern*, poiché implica la necessità di fornire un'interfaccia tale da soddisfare l'istruzione `foreach`. Quando disponete di un'interfaccia e ve ne occorre un'altra potete realizzare un adaptatore. In questo esempio l'autore ha voluto fornire la possibilità di produrre un iteratore inverso (*reverse iterator*), in aggiunta a quello standard (*forward iterator*), pertanto non ha potuto ricorrere alla tecnica di sovrascrittura; ha quindi aggiunto un metodo che produce un oggetto **Iterable** utilizzabile nell'istruzione `foreach`. Come potete vedere questa soluzione consente di utilizzare `foreach` in diversi modi.

```
//: holding/AdapterMethodIdiom.java
// L'Adapter Method idiom permette di utilizzare foreach
// con altri tipi di oggetti Iterable.
import java.util.*;

class ReversibleArrayList<T> extends ArrayList<T> {
    public ReversibleArrayList(Collection<T> c) { super(c); }
    public Iterable<T> reversed() {
        return new Iterable<T>() {
            public Iterator<T> iterator() {
                return new Iterator<T>() {
                    int current = size() - 1;
                    public boolean hasNext() { return current > -1; }
                    public T next() { return get(current--); }
                    public void remove() { // Non implementato
                        throw new UnsupportedOperationException();
                    }
                };
            }
        };
    }
}
```



```
    };
}
};

public class AdapterMethodIdiom {
    public static void main(String[] args) {
        ReversibleArrayList<String> ral =
            new ReversibleArrayList<String>(
                Arrays.asList("To be or not to be".split(" ")));
        // Cattura l'iteratore normale tramite iterator():
        for(String s : ral)
            System.out.print(s + " ");
        System.out.println();
        // Gestisce l'Iterable desiderato
        for(String s : ral.reversed())
            System.out.print(s + " ");
    }
} /* Output:
To be or not to be
be to not or be To
*///:~
```

Specificando semplicemente l'oggetto **ral** nella dichiarazione `foreach` otterrete l'iteratore predefinito, che procede in avanti, ma se chiamate `reversed()` sull'oggetto produrrete un comportamento diverso.

Grazie a questo approccio è possibile aggiungere due metodi Adapter all'esempio di **IterableClass.java**.

```
//: holding/MultiIterableClass.java
// Aggiunta di diversi metodi Adapter.
import java.util.*;

public class MultiIterableClass extends IterableClass {
    public Iterable<String> reversed() {
        return new Iterable<String>() {
```



```

public Iterator<String> iterator() {
    return new Iterator<String>() {
        int current = words.length - 1;
        public boolean hasNext() { return current > -1; }
        public String next() { return words[current--]; }
        public void remove() { // Non implementato
            throw new UnsupportedOperationException();
        }
    };
}
public Iterable<String> randomized() {
    return new Iterable<String>() {
        public Iterator<String> iterator() {
            List<String> shuffled =
                new ArrayList<String>(Arrays.asList(words));
            Collections.shuffle(shuffled, new Random(47));
            return shuffled.iterator();
        }
    };
}
public static void main(String[] args) {
    MultiIterableClass mic = new MultiIterableClass();
    for(String s : mic.reversed())
        System.out.print(s + " ");
    System.out.println();
    for(String s : mic.randomized())
        System.out.print(s + " ");
    System.out.println();
    for(String s : mic)
        System.out.print(s + " ");
    }
} /* Output:
banana-shaped. be to Earth the know we how is that And
is banana-shaped. Earth that how the be And we know to

```



And that is how we know the Earth to be banana-shaped.

*/~

Noteate come il secondo metodo, **random()**, non generi il proprio **Iterator** ma si limiti a restituire quello della **List** mescolata. Potete osservare dall'output che il metodo **Collections.shuffle()** non altera l'array originale, bensì mescola soltanto i riferimenti in **shuffled**. Questo avviene soltanto perché il metodo **randomized()** ingloba in un **ArrayList** il risultato di **Arrays.asList()**. Se la **List** prodotta da **Arrays.asList()** viene mescolata direttamente modificherà l'array sottostante, come mostrato dal seguente esempio.

```

//: holding/ModifyingArraysAsList.java
import java.util.*;

public class ModifyingArraysAsList {
    public static void main(String[] args) {
        Random rand = new Random(47);
        Integer[] ia = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        List<Integer> list1 =
            new ArrayList<Integer>(Arrays.asList(ia));
        System.out.println("Before shuffling: " + list1);
        Collections.shuffle(list1, rand);
        System.out.println("After shuffling: " + list1);
        System.out.println("array: " + Arrays.toString(ia));

        List<Integer> list2 = Arrays.asList(ia);
        System.out.println("Before shuffling: " + list2);
        Collections.shuffle(list2, rand);
        System.out.println("After shuffling: " + list2);
        System.out.println("array: " + Arrays.toString(ia));
    }
} /* Output:
Before shuffling: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
After shuffling: [4, 6, 3, 1, 8, 7, 2, 5, 10, 9]
array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Before shuffling: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
After shuffling: [9, 1, 6, 3, 7, 2, 5, 10, 4, 8]

```



```
array: [9, 1, 6, 3, 7, 2, 5, 10, 4, 8]
*///:~
```

Nel primo caso l'output di `Arrays.asList()` è passato al costruttore di `ArrayList()` per creare un `ArrayList` che fa riferimento agli elementi di `ia`. Il mescolamento di questi riferimenti non modifica l'array, tuttavia, se utilizzate direttamente il risultato di `Arrays.asList(ia)`, il mescolamento altererà l'ordine di `ia`. È importante che siate consapevoli del fatto che `Arrays.asList()` produce un oggetto `List` che si serve dell'array sottostante come propria implementazione fisica. Se la `List` è sottoposta a qualsiasi trattamento che la modifichi e se non volete che l'array originale venga ugualmente modificato, dovreste creare una copia della lista in un altro contenitore.

Esercizio 32 (2) Seguendo l'esempio di `MultIterableClass` aggiungete i metodi `reversed()` e `randomized()` a `NonCollectionSequence.java`, e fate in modo che `NonCollectionSequence` implementi `Iterable`. Dimostrate che queste tecniche funzionano con le istruzioni `foreach`.

Riepilogo

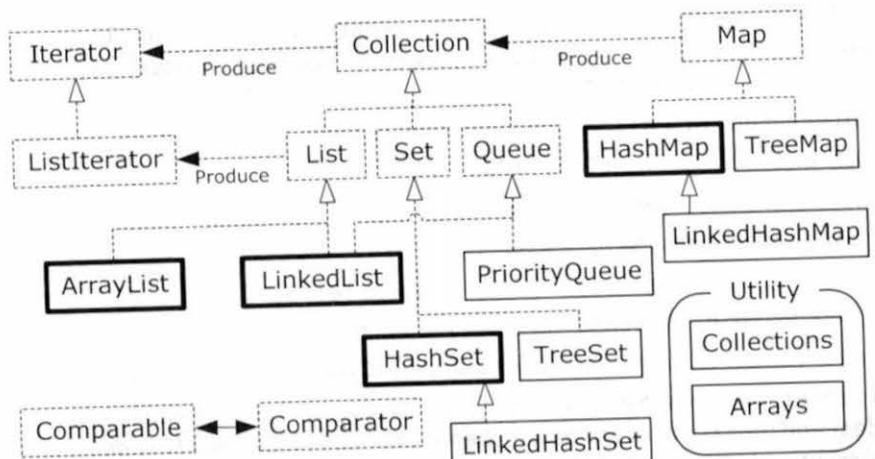
Java mette a disposizione diversi strumenti e modi per contenere gli oggetti.

1. Un array associa gli indici numerici agli oggetti, conservando gli oggetti di un tipo conosciuto in modo da non dover eseguire il cast sul risultato in fase di ricerca di uno di essi. L'array può essere multidimensionale e contenere primitivi, tuttavia le dimensioni non sono modificabili dopo la sua creazione.
2. Una **Collection** contiene elementi singoli, mentre una **Map** gestisce copie chiave/valore. Mediante i generici di Java specificate il tipo di oggetto che dovrà essere inserito nei contenitori, in modo da non poter inserirvi un tipo errato e da non dover sottoporre a cast gli elementi al momento del loro recupero dal contenitore. Sia **Collection** sia **Map** si ridimensionano automaticamente quando aggiungete altri elementi. Sebbene un contenitore non possa contenere tipi primitivi, la funzionalità di autoboxing si incarica di convertire i primitivi nei rispettivi tipi wrapper gestiti dal contenitore e viceversa.
3. Come gli array, anche le **List** associano gli indici numerici agli oggetti: per questo motivo entrambi sono considerati contenitori ordinati.
4. Se avete necessità di eseguire ripetutamente accessi casuali, utilizzate un **ArrayList**; se invece eseguite inserimenti e rimozioni nel mezzo di un elenco, servitevi preferibilmente di una **LinkedList**.



5. Il comportamento delle **Queue** (code) e degli **stack** (pile) è definito tramite **LinkedList**.
6. Una **Map** è una tecnica per associare oggetti ad altri oggetti, non a valori interi. Le **HashMap** sono state progettate per consentire un accesso rapido; una **TreeMap**, di contro, mantiene ordinate le proprie chiavi e per questa ragione non è veloce quanto una **HashMap**. Una **LinkedHashMap** conserva gli elementi in ordine d'inserimento, tuttavia garantisce un accesso rapido mediante il cosiddetto *hashing*.
7. Un **Set** accetta soltanto elementi univoci. Gli **HashSet** offrono la massima rapidità nell'esecuzione delle ricerche, i **TreeSet** mantengono ordinati gli elementi, mentre i **LinkedHashSet** li conservano in ordine d'inserimento.
8. Non vi è alcuna necessità di utilizzare le vecchie classi **Vector**, **Hashtable** e **Stack** nel nuovo codice.

Si è ritenuto opportuno fornire un diagramma semplificato dei contenitori Java, senza tenere conto delle classi astratte né dei vecchi componenti: il diagramma seguente include le sole interfacce e classi che utilizzerete regolarmente.



Tassonomia semplificata dei contenitori Java

Notate che esistono soltanto quattro contenitori di base (**Map**, **List**, **Set** e **Queue**) e soltanto due o tre implementazioni di ciascuno: le implementazioni **java.util.concurrent** di **Queue** non figurano in questo diagramma. I contenitori che utilizzerete con maggiore frequenza sono indicati dal bordo nero più spesso.

I riquadri con il bordo tratteggiato rappresentano le interfacce, mentre quelli con il bordo continuo indicano le normali classi "concrete". Le frecce tratteggiate con la punta bianca indicano che una determinata classe implementa un'interfaccia; di contro, le frecce con la punta nera segnalano che una classe può produrre oggetti della classe indicata dalla freccia. Per esempio, qualsiasi **Collection** può generare un **Iterator** e una **List** può produrre un **ListIterator** (oltre a un normale **Iterator**, dal momento che la **List** viene ereditata da **Collection**).

Di seguito è mostrato un esempio che evidenzia la differenza nei metodi delle diverse classi; il codice effettivo è riportato nel Volume 2, Capitolo 3 mentre in questa sede viene elencato soltanto l'output, per mostrare le interfacce implementate in ogni classe o interfaccia.

```
//: holding/ContainerMethods.java
import net.mindview.util.*;

public class ContainerMethods {
    public static void main(String[] args) {
        ContainerMethodDifferences.main(args);
    }
} /* Output: (Sample)
Collection: [add, addAll, clear, contains, containsAll,
equals, hashCode, isEmpty, iterator, remove, removeAll,
retainAll, size, toArray]
Interfaces in Collection: [Iterable]
Set extends Collection, adds: []
Interfaces in Set: [Collection]
HashSet extends Set, adds: []
Interfaces in HashSet: [Set, Cloneable, Serializable]
LinkedHashSet extends HashSet, adds: []
Interfaces in LinkedHashSet: [Set, Cloneable, Serializable]
TreeSet extends Set, adds: [pollLast, navigableHeadSet,
descendingIterator, lower, headSet, ceiling, pollFirst,
subSet, navigableTailSet, comparator, first, floor, last,
navigableSubSet, higher, tailSet]
```

Interfaces in TreeSet: [NavigableSet, Cloneable, Serializable]
List extends Collection, adds: [listIterator, indexOf, get,
subList, set, lastIndexOf]
Interfaces in List: [Collection]
ArrayList extends List, adds: [ensureCapacity, trimToSize]
Interfaces in ArrayList: [List, RandomAccess, Cloneable,
Serializable]
LinkedList extends List, adds: [pollLast, offer,
descendingIterator, addFirst, peekLast, removeFirst,
peekFirst, removeLast, getLast, pollFirst, pop, poll, addLast,
removeFirstOccurrence, getFirst, element, peek, offerLast,
push, offerFirst, removeLastOccurrence]
Interfaces in LinkedList: [List, Deque, Cloneable,
Serializable]
Queue extends Collection, adds: [offer, element, peek, poll]
Interfaces in Queue: [Collection]
PriorityQueue extends Queue, adds: [comparator]
Interfaces in PriorityQueue: [Serializable]
Map: [clear, containsKey, containsValue, entrySet, equals, get,
hashCode, isEmpty, keySet, put, putAll, remove, size, values]
HashMap extends Map, adds: []
Interfaces in HashMap: [Map, Cloneable, Serializable]
LinkedHashMap extends HashMap, adds: []
Interfaces in LinkedHashMap: [Map]
SortedMap extends Map, adds: [subMap, comparator, firstKey,
lastKey, headMap, tailMap]
Interfaces in SortedMap: [Map]
TreeMap extends Map, adds: [descendingEntrySet, subMap,
pollLastEntry, lastKey, floorEntry, lastEntry, lowerKey,
navigableHeadMap, navigableTailMap, descendingKeySet, tailMap,
ceilingEntry, higherKey, pollFirstEntry, comparator, firstKey,
floorKey, higherEntry, firstEntry, navigableSubMap, headMap,
lowerEntry, ceilingKey]
Interfaces in TreeMap: [NavigableMap, Cloneable, Serializable]
*/~:

Potete notare che, tranne **TreeSet**, tutti i **Set** hanno esattamente la stessa interfaccia di **Collection**. **List** e **Collection** presentano differenze sostanziali, benché **List** richieda metodi che sono disponibili in **Collection**. D'altra parte, i metodi nell'interfaccia **Queue** sono autonomi: non è necessario disporre



Stringhe

dei metodi di **Collection** per creare un'implementazione **Queue** funzionante. Infine, l'unico punto in comune tra **Map** e **Collection** è il fatto che una **Map** può produrre alcune **Collection** utilizzando i metodi **entrySet()** e **values()**.

Osservate il modo in cui l'interfaccia **java.util.RandomAccess** appare nel diagramma: è collegata ad **ArrayList** ma non a **LinkedList** e fornisce informazioni agli algoritmi che potrebbero modificare il loro comportamento a seconda dell'utilizzo di una particolare **List**.

È vero che questa struttura organizzativa è insolita rispetto alle gerarchie orientate agli oggetti, ma via via che apprenderete altri dettagli sui contenitori in **java.util**, in particolare nel Volume 2, Capitolo 5, vi renderete conto che esistono problemi ben più seri di una semplice "stranezza" nella struttura ereditaria. La progettazione delle librerie di contenitori ha sempre presentato problemi complessi, la cui soluzione richiede un insieme di forze che spesso si contrastano vicendevolmente: preparatevi quindi a far fronte ad alcuni compromessi.

Malgrado questi problemi, i contenitori Java sono strumenti fondamentali che potrete utilizzare nella vostra attività quotidiana per rendere i vostri programmi più semplici, potenti ed efficaci.

Vi occorrerà tempo per acquisire familiarità con alcuni aspetti della libreria, tuttavia con un po' di pazienza vedrete che l'utilizzo delle classi non sarà un'impresa insormontabile.

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



La manipolazione delle stringhe è senza dubbio una delle attività più comuni in programmazione.

Questa considerazione è valida soprattutto per i programmi web, ambiente in cui Java trova larga applicazione. In questo capitolo potrete esaminare in modo approfondito la classe innegabilmente più utilizzata in questo linguaggio, **String**, nonché alcune tra le classi e utility a essa associati.

Stringhe invariabili

Gli oggetti della classe **String** sono immutabili. Se consultate la documentazione JDK per questa classe vedrete che ogni metodo che sembra modificare una **String**, in realtà, genera e restituisce un oggetto **String** ex-novo contenente la modifica richiesta: la **String** originale resta intatta.

Considerate il seguente codice.

```
//: strings/Immutable.java
import static net.mindview.util.Print.*;
```



```

public class Immutable {
    public static String upcase(String s) {
        return s.toUpperCase();
    }
    public static void main(String[] args) {
        String q = "howdy!";
        print(q); // howdy!
        String qq = upcase(q);
        print(qq); // HOWDY!
        print(q); // howdy!
    }
} /* Output:
howdy!
HOWDY!
howdy!
*///:~

```

La variabile **q** passata a **upcase()**, in realtà, è una copia del riferimento a **q**. L'oggetto al quale è collegato questo riferimento si trova in una sola ubicazione fisica, mentre i riferimenti sono copiati via via che vengono passati.

Esaminando la definizione di **upcase()** potete vedere che il riferimento che viene passato è chiamato **s**, ed esiste soltanto finché il corpo di **upcase()** rimane in esecuzione. Al completamento di **upcase()** il riferimento locale **s** scompare e **upcase()** restituisce il risultato, che corrisponde alla stringa originale con tutti i caratteri in maiuscolo. Naturalmente in realtà Java restituisce un riferimento al risultato, ma si dà il caso che il riferimento restituito sia un nuovo oggetto e che l'oggetto **q** originale sia rimasto invariato.

Questo comportamento è generalmente quello che vi auspicate. Supponete di scrivere:

```

String s = "asdf";
String x = Immutable.upcase(s);

```

Volete davvero che il metodo **upcase()** alteri l'argomento? Al lettore del codice, un argomento appare generalmente come un'informazione fornita al metodo, non come qualcosa che possa essere modificato. Questa è una garanzia importante, poiché rende il codice più facile da scrivere e comprendere.



Confronto tra sovraccarico di “+” e StringBuilder

Dal momento che gli oggetti **String** sono invariabili, potete creare un alias di una determinata **String** quante volte desiderate. Poiché una **String** è di sola lettura, è impossibile che un riferimento cambi qualcosa che coinvolga gli altri riferimenti.

L'invariabilità può dare luogo a problemi di efficienza. Un esempio calzante è rappresentato dall'operatore “+”, che per gli oggetti **String** è stato sovraccaricato: ricordate che il sovraccarico (*overloading*) significa che a un'operazione viene assegnato un ulteriore significato nell'ambito di una classe particolare. In Java gli operatori “+” e “+=” sono gli unici a essere sovraccaricati.¹

L'operatore “+” permette di concatenare le **String**.

```

//: strings/Concatenation.java

public class Concatenation {
    public static void main(String[] args) {
        String mango = "mango";
        String s = "abc" + mango + "def" + 47;
        System.out.println(s);
    }
} /* Output:
abcmangodef47
*///:~

```

Non è difficile immaginare come un programma del genere potrebbe funzionare. La **String** “abc” potrebbe disporre di un metodo **append()** che genera un nuovo oggetto **String** contenente la stringa stessa concatenata con il contenuto di **mango**. Il nuovo oggetto **String** genererebbe a quel punto un'altra nuova **String** che aggiungerebbe “def” e così via.

1. C++ permette ai programmati di sovraccaricare gli operatori a loro discrezione. Poiché spesso si tratta di un procedimento complesso (si veda il Capitolo 10 di *Thinking in C++*, 2a edizione, Prentice Hall, 2000), i progettisti di Java hanno ritenuto che questa funzionalità non dovesse essere inclusa in Java. Non è stato poi così male che alla fine si riducessero a farlo loro stessi e, ironia della sorte, è curioso notare che il sovraccarico dell'operatore sarebbe stato molto più facile da utilizzare in Java di quanto non lo fosse in C++. Questa caratteristica è osservabile in Python (www.python.org) e in C#, che dispongono di una funzionalità immediata di sovraccarico degli operatori.



Questo meccanismo funzionerebbe senza problemi, ma richiederebbe la creazione di molti oggetti **String** soltanto per assemblare la nuova **String**: a quel punto vi ritrovereste con una grande quantità di oggetti **String** intermedi che dovrebbero rimanere in attesa della garbage collection. Forse i progettisti Java hanno provato in primo luogo questa tecnica, che è una classica lezione nella progettazione software: non si sa realmente nulla di un software fino a quando non si prova a scrivere codice ottenendo qualcosa che funziona. Probabilmente i progettisti si sono resi conto che questo meccanismo era inaccettabile in termini di prestazioni.

Per vedere ciò che accade realmente potete decompilare il codice precedente per mezzo dell'utility **javap**, un componente di JDK, direttamente dalla riga di comando:

```
javap -c Concatenation
```

L'opzione **-c** produrrà il bytecode della JVM. Dopo avere editato il codice per eliminare le parti non determinanti ai fini di questo esempio, otterrete i bytecode relativi.

```
public static void main(java.lang.String[]);
Code:
Stack=2, Locals=3, Args_size=1
0: ldc #2; //String mango
2: astore_1
3: new #3; //class StringBuilder
6: dup
7: invokespecial #4; //StringBuilder. "<init>":()
10: ldc #5; //String abc
12: invokevirtual #6; //StringBuilder.append:(String)
15: aload_1
16: invokevirtual #6; //StringBuilder.append:(String)
19: ldc #7; //String def
21: invokevirtual #6; //StringBuilder.append:(String)
24: bipush 47
26: invokevirtual #8; //StringBuilder.append:(I)
29: invokevirtual #9; //StringBuilder.toString:()
32: astore_2
33: getstatic #10; //Field System.out:PrintStream;
```



```
36: aload_2
```

```
37: invokevirtual #11; // PrintStream.println:(String)
```

```
40: return
```

Se avete esperienza con il linguaggio Assembler questo codice potrà sembrarvi familiare, visto che istruzioni quali **dup** e **invokevirtual** sono l'equivalente JVM del linguaggio Assembler. Se invece non avete mai visto nulla in Assembler, non preoccupatevi: ciò che è importante notare è l'utilizzo da parte del compilatore della classe **java.lang.StringBuilder**. Il codice sorgente originale non faceva cenno a **StringBuilder**, tuttavia il compilatore ha comunque deciso di servirsene poiché molto più efficiente.

In questo caso il compilatore crea un oggetto **StringBuilder** per costruire la **String s** e chiama **append()** quattro volte, una per ciascuno degli elementi che compongono la stringa finale (*abcmanodef47*); al termine, chiama **toString()** per produrre il risultato, che registra come **s** con **astore_2**.

Prima che iniziiate a pensare di dovere utilizzare le **String** ovunque e che il compilatore si incaricherà di ottimizzare al meglio tutto il vostro codice, è opportuno che esaminiate con maggiore attenzione l'attività del compilatore. Considerate l'esempio seguente, che produce un risultato di tipo **String** in due modi, utilizzando le **String** e tramite l'utilizzo manuale di **StringBuilder**.

```
//: strings/WhitherStringBuilder.java

public class WhitherStringBuilder {
    public String implicit(String[] fields) {
        String result = "";
        for(int i = 0; i < fields.length; i++)
            result += fields[i];
        return result;
    }
    public String explicit(String[] fields) {
        StringBuilder result = new StringBuilder();
        for(int i = 0; i < fields.length; i++)
            result.append(fields[i]);
        return result.toString();
    }
}
```

A questo punto, se eseguite **javap -c WitherStringBuilder** noterete il codice (semplificato) per i due metodi. In primo luogo, **implicit()**:

```
public java.lang.String implicit(java.lang.String[]);
Code:
0: ldc #2; //String
2: astore_2
3: iconst_0
4: istore_3
5: iload_3
6: aload_1
7: arraylength
8: if_icmpge 38
11: new #3; //class StringBuilder
14: dup
15: invokespecial #4; // StringBuilder. "<init>":()
18: aload_2
19: invokevirtual #5; // StringBuilder.append:()
22: aload_1
23: iload_3
24: aaload
25: invokevirtual #5; // StringBuilder.append:()
28: invokevirtual #6; // StringBuilder.toString:()
31: astore_2
32: iinc 3, 1
35: goto 5
38: aload_2
39: areturn
```

Osservate le righe **8:** e **35:** che insieme formano un'iterazione. **8:** esegue un "confronto maggiore di-uguale a di tipo Integer" (**if_icmpge**, *Integer Compare Greater than or Equal to*) tra gli operandi in stack, ovvero verifica se il primo valore **Integer** è maggiore o uguale al secondo, poi salta a **38:** a completamento del ciclo. **35:** è un'istruzione per ritornare di nuovo all'inizio del ciclo, a **5:**. L'elemento importante da considerare è che la costruzione di **StringBuilder** avviene all'interno di questa iterazione: questo significa che ottenete un nuovo oggetto **StringBuilder** ogni volta che passate attraverso il ciclo.

Di seguito è mostrato il bytecode per **explicit()**:

```
public java.lang.String explicit(java.lang.String[]);
Code:
0: new #3; //class StringBuilder
3: dup
4: invokespecial #4; // StringBuilder. "<init>":()
7: astore_2
8: iconst_0
9: istore_3
10: iload_3
11: aload_1
12: arraylength
13: if_icmpge 30
16: aload_2
17: aload_1
18: iload_3
19: aaload
20: invokevirtual #5; // StringBuilder.append:()
23: pop
24: iinc 3, 1
27: goto 10
30: aload_2
31: invokevirtual #6; // StringBuilder.toString:()
34: areturn
```

Non soltanto il codice del ciclo è più corto e più semplice, ma il metodo genera un solo oggetto **StringBuilder**. La creazione di uno **StringBuilder** esplicito consente anche di preassegnarne le dimensioni, se sapete in anticipo quali potrebbero essere, in modo tale che l'oggetto non debba riallocare continuamente il buffer.

Per questo motivo, quando generate un metodo **toString()**, se le operazioni sono abbastanza semplici da essere risolte dal compilatore in autonomia, di norma potrete contare sul fatto che questo svilupperà i risultati in modo ragionevole.



Tuttavia se il procedimento implica un'iterazione dovreste utilizzare esplicitamente uno **StringBuilder** nel vostro metodo **toString()**, come nell'esempio seguente.

```
//: strings/UsingStringBuilder.java
import java.util.*;

public class UsingStringBuilder {
    public static Random rand = new Random(47);
    public String toString() {
        StringBuilder result = new StringBuilder("[");
        for(int i = 0; i < 25; i++) {
            result.append(rand.nextInt(100));
            result.append(",");
        }
        result.delete(result.length()-2, result.length());
        result.append("]");
        return result.toString();
    }
    public static void main(String[] args) {
        UsingStringBuilder usb = new UsingStringBuilder();
        System.out.println(usb);
    }
} /* Output:
[58, 55, 93, 61, 61, 29, 68, 0, 22, 7, 88, 28, 51, 89, 9,
78, 98, 61, 20, 58, 16, 40, 11, 22, 4]
*///:~
```

Osservate come ogni porzione del risultato venga aggiunta mediante una dichiarazione **append()**. Se provate a ricorrere a scorciatoie con qualcosa di simile ad **append(a + ":" + c)**, il compilatore inizierà di nuovo a creare altri oggetti **StringBuilder**.

Se non siete sicuri dell'approccio da adottare, potete eseguire **javap** per un ulteriore controllo.

Sebbene **StringBuilder** disponga di un completo assortimento di metodi, tra cui **insert()**, **replace()**, **substring()** e persino **reverse()**, quelli che utilizzerete regolarmente sono **append()** e **toString()**. Tenete presente anche l'utilizzo di



delete() per rimuovere l'ultima virgola e lo spazio prima di aggiungere la parentesi quadra di chiusura.

La classe **StringBuilder** è stata introdotta con Java SE5: prima di essa si utilizzava **StringBuffer**, che garantendo la massima sicurezza nella gestione dei thread (si veda il Volume 3, Capitolo 1) è di utilizzo decisamente oneroso. Di conseguenza, l'esecuzione delle operazioni di stringa in Java SE5/6 dovrebbe risultare più veloce.

Esercizio 1 (2) Analizzate **SprinklerSystem.toString()** in **reusing/SprinklerSystem.java** per determinare se scrivendo il metodo **toString()** con uno **StringBuilder** esplicito eviterete di creare troppi **StringBuilder**.

Ricorrenza non intenzionale

Come qualsiasi altra classe i contenitori standard di Java ereditano da **Object**, pertanto tutti possiedono un metodo **toString()**.

Questo metodo è stato sovrascritto in modo da produrre una rappresentazione **String** dei contenitori standard di Java, comprensiva degli oggetti che contengono. **ArrayList.toString()**, per esempio, considera i singoli elementi dell'**ArrayList** e per ciascuno di essi chiama **toString()**.

```
//: strings/ArrayListDisplay.java
import generics.coffee.*;
import java.util.*;

public class ArrayListDisplay {
    public static void main(String[] args) {
        ArrayList<Coffee> coffees = new ArrayList<Coffee>();
        for(Coffee c : new CoffeeGenerator(10))
            coffees.add(c);
        System.out.println(coffees);
    }
} /* Output:
[Americano 0, Latte 1, Americano 2, Mocha 3, Mocha 4, Breve
5, Americano 6, Latte 7, Cappuccino 8, Cappuccino 9]
*///:~
```

Supponete di volere fare in modo che `toString()` visualizzi l'indirizzo della vostra classe. Un semplice riferimento a `this` sembrerebbe sufficiente.

```
//: strings/InfiniteRecursion.java
// Ricorrenza accidentale.
// {RunByHand}
import java.util.*;

public class InfiniteRecursion {
    public String toString() {
        return "InfiniteRecursion address: " + this + "\n";
    }
    public static void main(String[] args) {
        List<InfiniteRecursion> v =
            new ArrayList<InfiniteRecursion>();
        for(int i = 0; i < 10; i++)
            v.add(new InfiniteRecursion());
        System.out.println(v);
    }
} //:~
```

Se generate un oggetto **InfiniteRecursion** e lo visualizzate, otterrete un lunghissimo elenco di eccezioni, come pure se disponete gli oggetti **InfiniteRecursion** in un **ArrayList** e la visualizzate, come mostrato nel codice.

Il motivo di questo comportamento è l'intervento della funzionalità di conversione automatica di tipo per **String**. Quando scrivete

```
"InfiniteRecursion address: " + this
```

il compilatore vede una **String** seguita da un "+" e da qualcosa che non è una **String**, quindi cerca di convertire `this` in **String**. Questa conversione viene eseguita chiamando `toString()`, e ciò produce una chiamata ricorsiva.

Se desiderate visualizzare l'indirizzo dell'oggetto, la soluzione consiste nel chiamare il metodo `toString()` di **Object**, che si occupa esattamente di questo. Pertanto, invece di scrivere `this`, utilizzerete `super.toString()`.

Esercizio 2 (1) Correggete il problema in **InfiniteRecursion.java**.

Operazioni sulle stringhe

Di seguito sono elencati alcuni metodi di base disponibili per gli oggetti **String**. I metodi sovraccarichi sono riportati nella stessa riga.

Metodo	Argomenti, Overloading	Utilizzo
Costruttore	Metodi sovraccarichi: predefinito, String , StringBuilder , StringBuffer , array di char , array di byte .	Generazione di oggetti String .
length()		Restituisce il numero di caratteri in String .
charAt()	Indice int	Restituisce il char presente all'interno di String nella posizione specificata.
getChars() , getBytes()	L'inizio e la fine da cui eseguire la copia, l'array in cui copiare, un indice nell'array di destinazione.	Copia char o byte in un array esterno.
toCharArray()		Produce un array char[] contenente i caratteri di String .
equals() , equals- IgnoreCase()	Una String da confrontare.	Controlla l'uguaglianza del contenuto di due String .
compareTo()	Una String da confrontare.	Restituisce un valore negativo, zero o un valore positivo in base all'ordinamento lessicografico della String e dell'argomento. Differenzia tra lettere maiuscole e minuscole.
contains()	Una CharSequence da ricercare.	Restituisce true se l'argomento è contenuto nella String .
contentEquals()	Una CharSequence o uno StringBuffer da confrontare.	Restituisce true se l'argomento corrisponde in modo esatto.
equalsIgnoreCase()	Una String da confrontare.	Restituisce come risultato true se i contenuti sono uguali, senza tenere conto di maiuscole e minuscole.

(segue)



(continua)

Metodo	Argomenti, Overloading	Utilizzo
<code>regionMatches()</code>	L'offset nella String corrente, l'altra String e il suo offset e la lunghezza da confrontare. La versione sovraccarica del metodo non tiene conto di maiuscole e minuscole.	Restituisce un valore boolean che indica se le due sottostringhe (<i>region</i>) rappresentano sequenze di caratteri equivalenti.
<code>startsWith()</code>	La String con cui potrebbe iniziare la String corrente. La versione sovraccarica consente di specificare un offset come argomento.	Restituisce un valore boolean che indica se la String inizia con l'argomento.
<code>endsWith()</code>	La String con cui potrebbe terminare la String corrente.	Restituisce un valore boolean che indica se l'argomento è un suffisso.
<code>indexOf(), lastIndexOf()</code>	Metodi sovraccarichi: char , char e indice iniziale, String , String e indice iniziale.	Restituisce -1 se l'argomento non viene trovato all'interno della String corrente; in caso contrario restituisce l'indice a cui inizia l'argomento. lastIndexOf() esegue la ricerca a partire dal fondo.
<code>substring()</code> (anche <code>subSequence()</code>)	Metodi sovraccarichi: indice iniziale; indice iniziale e indice finale.	Restituisce un nuovo oggetto String che contiene l'insieme di caratteri specificato.
<code>concat()</code>	La String da concatenare.	Restituisce un nuovo oggetto String che contiene i caratteri della String originale seguiti da quelli presenti nell'argomento.
<code>replace()</code>	Il vecchio carattere da cercare e quello nuovo da sostituire. Il metodo può anche sostituire una CharSequence con un'altra CharSequence .	Restituisce un nuovo oggetto String dopo la sostituzione. Utilizza la vecchia String se non viene trovata nessuna corrispondenza.

(segue)

(continua)

Metodo	Argomenti, Overloading	Utilizzo
<code>toUpperCase()</code> <code>toLowerCase()</code>		Restituisce un nuovo oggetto String in cui tutte le lettere sono trasformate in maiuscole o minuscole, rispettivamente. Utilizza la vecchia String se non deve essere eseguito nessun cambiamento.
<code>trim()</code>		Restituisce un nuovo oggetto String in cui gli spazi vuoti vengono rimossi da ogni estremità. Utilizza la vecchia String se non deve essere eseguito nessun cambiamento.
<code>valueOf()</code>	Metodi sovraccarichi: Object , char[] , char[] e offset e conteggio, boolean , char , int , long , float , double .	Restituisce una String che contiene una rappresentazione carattere (char) dell'argomento.
<code>intern()</code>		Produce un unico riferimento alla String per una sequenza univoca di caratteri.

Potete vedere che ogni metodo di **String** ha cura di restituire un nuovo oggetto **String** quando è necessario cambiarne il contenuto; se il contenuto non deve essere modificato, invece, il metodo si limiterà a restituire un riferimento alla **String** originale: questo comportamento consente di ridurre lo spazio di memorizzazione degli oggetti e gli oneri in termini di CPU. I metodi di **String** che coinvolgono le espressioni regolari verranno illustrati nel prossimo del capitolo.

Formattazione dell'output

Una delle caratteristiche attese da lungo tempo e finalmente resa disponibile in Java SE5 è la formattazione dell'output prodotta nello stile **printf()** di C, che non soltanto permette di semplificare il codice per l'output, ma offre agli sviluppatori Java il massimo controllo sulla formattazione e l'allineamento dell'output.²

² Mark Walsh ha collaborato alla stesura di questo paragrafo e del paragrafo "Scansione



printf()

Il metodo **printf()** di C non assembla le stringhe allo stesso modo di Java ma accetta una stringa di formattazione e vi inserisce i valori passati come parametri, come nell'esempio seguente, formattandoli progressivamente. Invece di utilizzare l'operatore "+" sovraccarico (operazione non ammessa in C) per concatenare il testo quotato e le variabili, **printf()** ricorre a speciali segnaposto che indicano la posizione dei dati. Gli argomenti da inserire nella stringa di formattazione seguono in un elenco separato da virgole.

Per esempio:

```
printf("Row 1: [%d %f]\n", x, y);
```

Durante l'esecuzione il valore di **x** viene inserito in **%d** e quello di **y** in **%f**. Questi segnaposto sono chiamati modificatori di formato (o specificatori di formato, *format specifier*, secondo la terminologia classica di C): oltre a segnalare dove inserire il valore, indicano quale tipo di variabile deve essere inserito e come formattarlo. Per esempio, la "**%d**" della riga precedente indica che **x** è un numero intero, e "**%f**" che **y** è un valore a virgola mobile, **float** o **double**.

System.out.format()

Java SE5 ha introdotto il metodo **format()**, disponibile agli oggetti **PrintWriter** o **PrintStream** (si veda il Volume 2, Capitolo 6), che consentono di utilizzare **System.out** (*standard output*). Il metodo **format()** ricalca esattamente il **printf()** di C: i nostalgici di questo linguaggio hanno a disposizione persino il metodo di comodo **printf()**, che si limita a chiamare **format()**. Di seguito è mostrato un semplice esempio.

```
//: strings/SimpleFormat.java

public class SimpleFormat {
    public static void main(String[] args) {
        int x = 5;
        double y = 5.332542;
        // Il vecchio modo:
        System.out.println("Row 1: [" + x + " " + y + "]");
        // Il nuovo modo:
        System.out.format("Row 1: [%d %f]\n", x, y);
    }
}
```

```
// oppure
System.out.printf("Row 1: [%d %f]\n", x, y);
}
} /* Output:
Row 1: [5 5.332542]
Row 1: [5 5.332542]
Row 1: [5 5.332542]
*///:~
```

Come potete notare, **format()** e **printf()** sono equivalenti: in entrambi i casi è prevista una sola stringa di formattazione, seguita da un argomento per ogni modificatore di formato.

Classe Formatter

Tutte le nuove funzionalità di Java sono gestite dalla classe **Formatter** nel pacchetto **java.util**: questa classe è una sorta di traduttore che converte le vostre stringhe di formattazione e i dati nel risultato voluto. Quando generate un oggetto **Formatter** gli ordinate dove inserire questo risultato, comunicando tale informazione al costruttore.

```
//: strings/Turtle.java
import java.io.*;
import java.util.*;

public class Turtle {
    private String name;
    private Formatter f;
    public Turtle(String name, Formatter f) {
        this.name = name;
        this.f = f;
    }
    public void move(int x, int y) {
        f.format("%s The Turtle is at (%d,%d)\n", name, x, y);
    }
    public static void main(String[] args) {
        PrintStream outAlias = System.out;
        Turtle tommy = new Turtle("Tommy",
            new Formatter(outAlias)));
        tommy.move(5, 5);
    }
}
```

```

new Formatter(System.out));
Turtle terry = new Turtle("Terry",
    new Formatter(outAlias));
tommy.move(0,0);
terry.move(4,8);
tommy.move(3,4);
terry.move(2,5);
tommy.move(3,3);
terry.move(3,3);
}
} /* Output:
Tommy The Turtle is at (0,0)
Terry The Turtle is at (4,8)
Tommy The Turtle is at (3,4)
Terry The Turtle is at (2,5)
Tommy The Turtle is at (3,3)
Terry The Turtle is at (3,3)
*///:~

```

Tutto l'output di **tommy** è destinato a **System.out** e quello di **terry** a un alias di **System.out**. Il costruttore è sovraccaricato per tenere conto di un certo numero di posizioni di output: le versioni più utili sono le classi **PrintStream** (utilizzata nell'esempio), **OutputStream** e **File**. Vedrete meglio questo argomento nel Volume 2, Capitolo 6.

Esercizio 3 (1) Modificate **Turtle.java** in modo da inviare l'output a **System.err**.

L'esempio precedente utilizza un nuovo modificatore di formato, “%**s**”: esso indica un argomento **String** ed è un esempio del modificatore più semplice, che corrisponde a un solo tipo di conversione.

Modificatori di formato

Per controllare la spaziatura e l'allineamento in fase di inserimento dei dati si deve ricorrere a specificatori di formato più elaborati, secondo la sintassi generale mostrata di seguito.

%[argument_index\$][flags][width][.precision]conversion

Spesso è necessario controllare le dimensioni minime di un campo: per fare questo è possibile specificare un valore **width**. La classe **Formatter** garantisce che le dimensioni di un campo equivalgano a un determinato numero di caratteri, riempiendolo eventualmente di spazi vuoti. In modalità predefinita i dati sono allineati a destra, ma questo comportamento è modificabile includendo un “-” nella sezione **flags**.

L'opposto della larghezza è la precisione (**precision**), utilizzata per specificare un valore massimo. A differenza della larghezza, che è applicabile a tutti i tipi di conversione dati e per tutti si comporta allo stesso modo, la precisione ha un significato diverso in funzione dei tipi. Per le **String** specifica il numero massimo di caratteri che devono essere visualizzati; per i numeri in virgola mobile indica il numero di posizioni decimali da visualizzare (predefinite in 6), eseguendo un arrotondamento in caso di posizioni decimali eccessive o accodando zeri se sono insufficienti. Dal momento che i numeri interi non hanno componente decimale, a essi non si applica la precisione; pertanto, se utilizzate la precisione in una conversione di numeri interi solleverete un'eccezione. Questo esempio si serve dei modificatori di formato per visualizzare una ricevuta di acquisto.

```

//: strings/Receipt.java
import java.util.*;

public class Receipt {
    private double total = 0;
    private Formatter f = new Formatter(System.out);
    public void printTitle() {
        f.format("%-15s %5s %10s\n", "Item", "Qty", "Price");
        f.format("%-15s %5s %10s\n", "----", "----", "----");
    }
    public void print(String name, int qty, double price) {
        f.format("%-15.15s %5d %10.2f\n", name, qty, price);
        total += price;
    }
    public void printTotal() {
        f.format("%-15s %5s %10.2f\n", "Tax", "", total*0.06);
        f.format("%-15s %5s %10s\n", "", "", "----");
        f.format("%-15s %5s %10.2f\n", "Total", "",
            total * 1.06); }
    public static void main(String[] args) {

```

```

Receipt receipt = new Receipt();
receipt.printTitle();
receipt.print("Jack's Magic Beans", 4, 4.25);
receipt.print("Princess Peas", 3, 5.1);
receipt.print("Three Bears Porridge", 1, 14.29);
receipt.printTotal();
}

/* Output:
Item          Qty      Price
----          ---      ----
Jack's Magic Be    4      4,25
Princess Peas     3      5,10
Three Bears Por    1     14,29
Tax                  1,42
Total                25,06
*///:~

```

Come potete vedere, la classe **Formatter** garantisce un notevole controllo sulla spaziatura e l'allineamento per mezzo di una notazione ragionevolmente compatta. In questo caso le stringhe di formato sono semplicemente copiate per produrre la spaziatura richiesta.

Esercizio 4 (3) Modificate **Receipt.java** in modo che tutte le larghezze siano controllate da un unico insieme di valori costanti; l'obiettivo è permettervi di variare facilmente una larghezza cambiando un solo valore.

Conversioni di Formatter

Nella tabella seguente sono indicate le conversioni che avrete occasione di utilizzare più di frequente.

Caratteri di conversione	
d	Intero (come valore decimale)
c	Carattere Unicode
b	Valore booleano
s	Stringa
f	Virgola mobile (come valore decimale)

Caratteri di conversione

e	Virgola mobile (in notazione scientifica)
x	Intero (come valore esadecimale)
h	Codice hash (come valore esadecimale)
%	"%" così com'è

Questo esempio mostra le varie conversioni in azione.

```

//: strings/Conversion.java
import java.math.*;
import java.util.*;
public class Conversion {
    public static void main(String[] args) {
        Formatter f = new Formatter(System.out);

        char u = 'a';
        System.out.println("u = 'a'");
        f.format("s: %s\n", u);
        // f.format("d: %d\n", u);
        f.format("c: %c\n", u);
        f.format("b: %b\n", u);
        // f.format("f: %f\n", u);
        // f.format("e: %e\n", u);
        // f.format("x: %x\n", u);
        f.format("h: %h\n", u);

        int v = 121;
        System.out.println("v = 121");
        f.format("d: %d\n", v);
        f.format("c: %c\n", v);
        f.format("b: %b\n", v);
        f.format("s: %s\n", v);
        // f.format("f: %f\n", v);
        // f.format("e: %e\n", v);
        f.format("x: %x\n", v);
        f.format("h: %h\n", v);
    }
}

```

```

BigInteger w = new BigInteger("5000000000000000");
System.out.println(
    "w = new BigInteger(\"5000000000000000\")");
f.format("d: %d\n", w);
// f.format("c: %c\n", w);
f.format("b: %b\n", w);
f.format("s: %s\n", w);
// f.format("f: %f\n", w);
// f.format("e: %e\n", w);
f.format("x: %x\n", w);
f.format("h: %h\n", w);

double x = 179.543;
System.out.println("x = 179.543");
// f.format("d: %d\n", x);
// f.format("c: %c\n", x);
f.format("b: %b\n", x);
f.format("s: %s\n", x);
f.format("f: %f\n", x);
f.format("e: %e\n", x);
// f.format("x: %x\n", x);
f.format("h: %h\n", x);

Conversion y = new Conversion();
System.out.println("y = new Conversion()");
// f.format("d: %d\n", y);
// f.format("c: %c\n", y);
f.format("b: %b\n", y);
f.format("s: %s\n", y);
// f.format("f: %f\n", y);
// f.format("e: %e\n", y);
// f.format("x: %x\n", y);
f.format("h: %h\n", y);
boolean z = false;
System.out.println("z = false");
// f.format("d: %d\n", z);

```

```

// f.format("c: %c\n", z);
f.format("b: %b\n", z);
f.format("s: %s\n", z);
// f.format("f: %f\n", z);
// f.format("e: %e\n", z);
// f.format("x: %x\n", z);
f.format("h: %h\n", z);

}
} /* Output: (Sample)
u = 'a'
s: a
c: a
b: true
h: 61
v = 121
d: 121
c: y
b: true
s: 121
x: 79
h: 79
w = new BigInteger("5000000000000000")
d: 5000000000000000
b: true
s: 5000000000000000
x: 2d79883d2000
h: 8842a1a7
x = 179.543
b: true
s: 179.543
f: 179.543000
e: 1.795430e+02
h: 1ef462c
y = new Conversion()
b: true
s: Conversion@9cab16

```

```

h: 9cab16
z = false
b: false
s: false
h: 4d5
*///:~

```

Le righe commentate si riferiscono a conversioni non valide per quel tipo particolare di variabile, la cui esecuzione produrrà un'eccezione.

Notate che il carattere di conversione '**b**' funziona per ciascuna delle suddette variabili, ma benché valido per qualsiasi tipo di argomento potrebbe non comportarsi come vi aspettereste. Per i primitivi **boolean** o gli oggetti **Boolean** il risultato sarà naturalmente **true** o **false**, tuttavia per qualsiasi altro argomento che non sia di tipo **null** il risultato sarà sempre **true**. Anche il valore numerico zero, che in molti linguaggi (tra cui il C) è sinonimo di **false**, produce **true**: pertanto prestate attenzione a utilizzare questa conversione con i tipi non booleani.

Esistono altri tipi di conversione meno noti e altre opzioni per i modificatori di formattazione, che troverete illustrati nella documentazione JDK per la classe **Formatter**.

Esercizio 5 (5) Per ognuno dei tipi di conversione elencati nella tabella precedente scrivete l'espressione di formattazione più complessa possibile, ossia quella che si serve di tutti i modificatori di formattazione disponibili per quel tipo di conversione.

String.format()

Java SE5 ha preso spunto anche dalla funzione **sprintf()** di C, impiegata nella creazione delle **String**. **String.format()** è un metodo **static** che, pur accettando gli stessi argomenti del metodo **format()** di **Formatter**, restituisce una **String**. Questo metodo può risultare pratico quando dovete chiamare **format()** una sola volta.

```

//: strings/DatabaseException.java

public class DatabaseException extends Exception {
    public DatabaseException(int transactionID, int queryID,
        String message) {
        super(String.format(" (t%d, q%d) %s", transactionID,
            queryID, message));
    }
}

```

```

    }
}

public static void main(String[] args) {
    try {
        throw new DatabaseException(3, 7, "Write failed");
    } catch(Exception e) {
        System.out.println(e);
    }
}

/* Output:
DatabaseException: (t3, q7) Write failed
*///:~

```

In realtà, accade semplicemente che **String.format()** istanzia una classe **Formatter** e le passa i vostri argomenti: l'utilizzo di questo metodo di comodo, tuttavia, può essere più chiaro e facile della stessa operazione eseguita manualmente.

Uno strumento per il dump esadecimale

Spesso si ha la necessità di esaminare un file binario in formato esadecimale; come secondo esempio, quindi, ecco un piccolo programma di utilità che visualizza un array binario di byte in un formato esadecimale comprensibile, per mezzo di **String.format()**.

```

//: net/mindview/util/Hex.java
package net.mindview.util;
import java.io.*;

public class Hex {
    public static String format(byte[] data) {
        StringBuilder result = new StringBuilder();
        int n = 0;
        for(byte b : data) {
            if(n % 16 == 0)
                result.append(String.format("%05X: ", n));
            result.append(String.format("%02X ", b));
            n++;
            if(n % 16 == 0) result.append("\n");
        }
        return result.toString();
    }
}

```



```

    }
    result.append("\n");
    return result.toString();
}

public static void main(String[] args) throws Exception {
    if(args.length == 0)
        // Test che visualizza in formato esadecimale
        // il file .class corrente:
        System.out.println(
            format(BinaryFile.read("Hex.class")));
    else
        System.out.println(
            format(BinaryFile.read(new File(args[0]))));
}
/* Output: (Sample)
00000: CA FE BA BE 00 00 00 31 00 52 0A 00 05 00 22 07
00010: 00 23 0A 00 02 00 22 08 00 24 07 00 25 0A 00 26
00020: 00 27 0A 00 28 00 29 0A 00 02 00 2A 08 00 2B 0A
00030: 00 2C 00 2D 08 00 2E 0A 00 02 00 2F 09 00 30 00
00040: 31 08 00 32 0A 00 33 00 34 0A 00 15 00 35 0A 00
00050: 36 00 37 07 00 38 0A 00 12 00 39 0A 00 33 00 3A
...
*/

```

Per aprire e leggere il file binario questo programma ricorre a un'altra utility che sarà introdotta nel Volume 2, Capitolo 6: **net.mindview.util.BinaryFile**. Il metodo **read()** restituisce l'intero file come array di **byte**.

Esercizio 6 (2) Create una classe contenente campi **int**, **long**, **float** e **double**. Per questa classe generate un metodo **toString()** che utilizzi **String.format()** e dimostrate che la vostra classe funziona correttamente.

Espressioni regolari

Le espressioni regolari sono state per lungo tempo confinate a programmi di utilità standard UNIX, come *sed* e *awk*, e a linguaggi quali Python e Perl; alcuni sostengono che a esse si deve gran parte del successo di Perl. In prece-



denza la manipolazione delle stringhe in Java era delegata alle classi **String**, **StringBuffer** e **StringTokenizer**, che offrivano strumenti relativamente più semplici delle espressioni regolari.

Le espressioni regolari sono strumenti potenti e flessibili per l'elaborazione dei testi, in senso letterale, naturalmente: esse consentono di specificare da programma modelli complessi di testo, i cosiddetti *pattern*, che possono essere ricercati in una stringa di input. Una volta che avete trovato la corrispondenza di questi modelli nei vostri testi, potete elaborare i risultati in qualsiasi modo desideriate. Nonostante la loro sintassi possa inizialmente intimorirvi, le espressioni regolari costituiscono un linguaggio compatto e dinamico che può essere impiegato per risolvere qualsiasi problema riguardi l'elaborazione delle stringhe, la corrispondenza e la selezione, l'editing e la verifica, in modo assolutamente generico.

Principi fondamentali

Un'espressione regolare o *regex*, è un modo per descrivere le stringhe in genere che potrebbe tradursi nell'espressione: "Se una stringa contiene questi elementi, allora fa al caso mio". Per esempio, se volete indicare che un numero potrebbe o non potrebbe essere preceduto da un segno meno, digitate questo segno seguito da un punto interrogativo, come in:

-?

Per descrivere un numero intero, scriverete che è composto di una o più cifre: nelle espressioni regolari una cifra viene descritta come "**\d**". Se avete esperienza con le espressioni regolari in altri linguaggi, noterete immediatamente una differenza nel modo in cui vengono gestite le barre inverse o *backslash*. In altri linguaggi, "****" significa: "Voglio inserire un normale backslash letterale nell'espressione regolare, senza assegnargli alcun significato speciale". In Java "****" significa invece "Sto inserendo un normale backslash di espressione, in modo che il carattere seguente abbia un significato speciale". Per esempio, la stringa di espressione regolare per indicare una cifra sarà "**\d**", mentre per inserire un backslash letterale scriverete "**\\d**". Tenete presente, tuttavia, che elementi come i simboli di nuova riga e le tabulazioni richiedono una sola barra inversa: "**\n\t**".

Per indicare "una o più occorrenze dell'espressione precedente" occorre utilizzare un "**+**"; per esempio, se volete ricercare "un possibile segno meno, seguito da una o più cifre" scriverete:

-?**\d+**



```

    }
    result.append("\n");
    return result.toString();
}

public static void main(String[] args) throws Exception {
    if(args.length == 0)
        // Test che visualizza in formato esadecimale
        // il file .class corrente:
        System.out.println(
            format(BinaryFile.read("Hex.class")));
    else
        System.out.println(
            format(BinaryFile.read(new File(args[0]))));
}
/* Output: (Sample)
00000: CA FE BA BE 00 00 00 31 00 52 0A 00 05 00 22 07
00010: 00 23 0A 00 02 00 22 08 00 24 07 00 25 0A 00 26
00020: 00 27 0A 00 28 00 29 0A 00 02 00 2A 08 00 2B 0A
00030: 00 2C 00 2D 08 00 2E 0A 00 02 00 2F 09 00 30 00
00040: 31 08 00 32 0A 00 33 00 34 0A 00 15 00 35 0A 00
00050: 36 00 37 07 00 38 0A 00 12 00 39 0A 00 33 00 3A
...
*/

```

Per aprire e leggere il file binario questo programma ricorre a un'altra utility che sarà introdotta nel Volume 2, Capitolo 6: **net.mindview.util.BinaryFile**. Il metodo **read()** restituisce l'intero file come array di **byte**.

Esercizio 6 (2) Create una classe contenente campi **int**, **long**, **float** e **double**. Per questa classe generate un metodo **toString()** che utilizzi **String.format()** e dimostrate che la vostra classe funziona correttamente.

Espressioni regolari

Le espressioni regolari sono state per lungo tempo confinate a programmi di utilità standard UNIX, come *sed* e *awk*, e a linguaggi quali Python e Perl; alcuni sostengono che a esse si deve gran parte del successo di Perl. In prece-



denza la manipolazione delle stringhe in Java era delegata alle classi **String**, **StringBuffer** e **StringTokenizer**, che offrivano strumenti relativamente più semplici delle espressioni regolari.

Le espressioni regolari sono strumenti potenti e flessibili per l'elaborazione dei testi, in senso letterale, naturalmente: esse consentono di specificare da programma modelli complessi di testo, i cosiddetti *pattern*, che possono essere ricercati in una stringa di input. Una volta che avete trovato la corrispondenza di questi modelli nei vostri testi, potete elaborare i risultati in qualsiasi modo desideriate. Nonostante la loro sintassi possa inizialmente intimorirvi, le espressioni regolari costituiscono un linguaggio compatto e dinamico che può essere impiegato per risolvere qualsiasi problema riguardi l'elaborazione delle stringhe, la corrispondenza e la selezione, l'editing e la verifica, in modo assolutamente generico.

Principi fondamentali

Un'espressione regolare o *regex*, è un modo per descrivere le stringhe in genere che potrebbe tradursi nell'espressione: "Se una stringa contiene questi elementi, allora fa al caso mio". Per esempio, se volete indicare che un numero potrebbe o non potrebbe essere preceduto da un segno meno, digitate questo segno seguito da un punto interrogativo, come in:

-?

Per descrivere un numero intero, scriverete che è composto di una o più cifre: nelle espressioni regolari una cifra viene descritta come "**\d**". Se avete esperienza con le espressioni regolari in altri linguaggi, noterete immediatamente una differenza nel modo in cui vengono gestite le barre inverse o *backslash*. In altri linguaggi, "****" significa: "Voglio inserire un normale backslash letterale nell'espressione regolare, senza assegnargli alcun significato speciale". In Java "****" significa invece "Sto inserendo un normale backslash di espressione, in modo che il carattere seguente abbia un significato speciale". Per esempio, la stringa di espressione regolare per indicare una cifra sarà "**\d**", mentre per inserire un backslash letterale scriverete "**\\d**". Tenete presente, tuttavia, che elementi come i simboli di nuova riga e le tabulazioni richiedono una sola barra inversa: "**\n\t**".

Per indicare "una o più occorrenze dell'espressione precedente" occorre utilizzare un "**+**"; per esempio, se volete ricercare "un possibile segno meno, seguito da una o più cifre" scriverete:

-?**\d+**



Il modo più semplice per utilizzare le espressioni regolari consiste nel servirsi della funzionalità nativa della classe **String**. Per esempio, ecco come verificare se una **String** corrisponde all'espressione regolare precedente.

```
//: strings/IntegerMatch.java
public class IntegerMatch {
    public static void main(String[] args) {
        System.out.println("-1234".matches("-?\\d+"));
        System.out.println("5678".matches("-?\\d+"));
        System.out.println("+911".matches("-?\\d+"));
        System.out.println("+911".matches("(-|\\+)?\\d+"));
    }
} /* Output:
true
true
false
true
*///:~
```

Le prime due espressioni corrispondono ma la terza inizia con un “+”, che pur essendo un simbolo legittimo significa che il numero non corrisponde all'espressione regolare. Occorre quindi un modo per dire “che può iniziare con + o con -”. Nelle espressioni regolari le parentesi hanno l'effetto di raggruppare l'espressione e la barra verticale () significa OR. Pertanto:

`(-|\\+)?`

vuol dire che questo componente della stringa può essere un “-”, un “+” oppure niente, a causa del “?”. Poiché nelle espressioni regolari il carattere “+” ha un significato speciale, deve essere sottoposto a escape mediante “\\” per comparire come carattere ordinario nell'espressione.

Una funzionalità utile in abbinamento alle espressioni regolari è il metodo **split()**, disponibile in **String**, che consente di separare la stringa in prossimità delle corrispondenze di una determinata espressione regolare.

```
//: strings/Splitting.java
import java.util.*;
```



```
public class Splitting {
    public static String knights =
        "Then, when you have found the shrubbery, you must" +
        "cut down the mightiest tree in the forest... " +
        "with... a herring! ";
    public static void split(String regex) {
        System.out.println(
            Arrays.toString(knights.split(regex)));
    }
    public static void main(String[] args) {
        split(""); // Non deve contenere caratteri di espressione
                   // regolare
        split("\\W+"); // Caratteri non ammessi nelle parole
        split("n\\W+"); // 'n' (nuova riga) seguita da caratteri
                       // non ammessi nelle parole
    }
} /* Output:
[Then,, when, you, have, found, the, shrubbery,, you, must,
cut, down, the, mightiest, tree, in, the, forest..., with..., 
a, herring!]
[Then, when, you, have, found, the, shrubbery, you, must, cut,
down, the, mightiest, tree, in, the, forest, with, a, herring]
[The, whe, you have found the shrubbery, you must cut dow, the
mightiest tree i, the forest... with... a herring!]
*///:~
```

Osservate innanzitutto che nelle espressioni regolari è possibile scrivere i normali caratteri: le espressioni non devono obbligatoriamente contenere caratteri speciali, come potete vedere nella prima chiamata a **split()** che esegue la suddivisione sugli spazi vuoti.

La seconda e terza chiamata a **split()** utilizzano “W”, che rappresenta un carattere di tipo *non-word*: notate, infatti, che nel secondo caso i segni di punteggiatura sono stati rimossi. Tenete presente che la versione minuscola, “w”, indica che si tratta di un carattere standard, normalmente utilizzato in informatica per comporre le comuni parole: A-Z, a-z, 0-9 e _ (simbolo di sottolineatura).³

3. La distinzione tra caratteri non-word e caratteri “di parola” (normali) può variare in funzione del supporto alle espressioni regolari fornito dal linguaggio: i caratteri A-Z e a-z

Nella terza chiamata a `split()` l'espressione indica “la lettera **n** seguita da uno o più caratteri non-word”: osservate infatti che gli schemi di suddivisione (*split pattern*) non appaiono nei risultati.

Una versione sovraccarica di `String.split()` consente di limitare il numero di suddivisioni cui devono essere soggette le stringhe. L'ultima funzionalità per le espressioni regolari disponibile in `String` è la sostituzione, che consente di sostituire la prima occorrenza oppure tutte.

```
//: strings/Replacing.java
import static net.mindview.util.Print.*;

public class Replacing {
    static String s = Splitting.knights;
    public static void main(String[] args) {
        print(s.replaceFirst("f\\w+", "located"));
        print(s.replaceAll("shrubbery|tree|herring", "banana"));
    }
} /* Output:
Then, when you have located the shrubbery, you must cut down
the mightiest tree in the forest... with... a herring!
Then, when you have found the banana, you must cut down the
mightiest banana in the forest... with... a banana!
*//*:~
```

La prima espressione esegue la corrispondenza per la lettera **f** seguita da uno o più caratteri “di parola” (**w** minuscola): cambia soltanto la prima corrispondenza trovata, pertanto sostituisce la parola “found” con “located”.

La seconda espressione cerca la corrispondenza di tre parole separate dalle barre verticali OR e sostituisce tutte le occorrenze trovate.

Avrete modo di scoprire che le espressioni regolari di tipo non `String` hanno funzionalità di sostituzione ben più potenti: per esempio, è possibile chiamare metodi per eseguire le sostituzioni.

Le espressioni regolari non `String` sono anche molto più efficienti nel caso dobbiate utilizzare più volte l'espressione regolare.

sono sempre inclusi, come generalmente anche i numeri e il simbolo di sottolineatura. Prima di distribuire il vostro programma, quindi, è sempre opportuno che eseguiate alcuni test di compatibilità.

Esercizio 7 (5) Basandovi sulla documentazione di `java.util.regex.Pattern`, scrivete e verificate un'espressione regolare per controllare che una frase cominci con una lettera maiuscola e termini con un punto.

Esercizio 8 (2) Eseguite la suddivisione della stringa `Splitting.knights` sulle parole “the” o “you”.

Esercizio 9 (4) Basandovi sulla documentazione di `java.util.regex.Pattern`, sostituite tutte le vocali in `Splitting.knights` con caratteri di sottolineatura.

Creazione di espressioni regolari

Potete cominciare ad accostarvi alle espressioni regolari lavorando con un sottoinsieme dei possibili costrutti. L'elenco completo dei costrutti ammessi per le espressioni regolari è disponibile nella documentazione JDK per la classe **Pattern** del pacchetto `java.util.regex`.

Caratteri	
B	Il carattere B specifico
\xhh	Carattere con valore esadecimale 0xhh
\uhhhh	Il carattere Unicode con la rappresentazione esadecimale 0xhhhh
\t	Tabulazione
\n	Nuova riga
\r	Ritorno di carrello (a capo)
\f	Avanzamento carta (<i>form feed</i>)
\e	Escape

La vera potenza delle espressioni regolari si avverte quando si definiscono le classi di carattere. Di seguito sono illustrati alcuni modi per generare queste classi e alcune classi predefinite.

Classi di carattere

.	Qualsiasi carattere
[abc]	Qualsiasi carattere tra a , b o c (equivale ad a b c)
[^abc]	Qualsiasi carattere tranne a , b e c (negazione)
[a-zA-Z]	Qualsiasi carattere da a a z o da A a Z (intervallo)
[abc hijl]	Qualsiasi carattere tra a,b,c,h,i,j o j (equivale ad a b c h i j) (unione)

Classi di carattere

[a-z&&[hij]]	Qualsiasi carattere tra h , i o j (intersezione)
\s	Uno spazio vuoto (spaziatura, tabulazione, nuova riga, avanzamento carta, ritorno di carrello)
\S	Un carattere diverso da uno spazio vuoto (^ \s)
\d	Un carattere numerico ([0-9])
\D	Un carattere non numerico ([^0-9])
\w	Un carattere di tipo word ([a-zA-Z_0-9])
\W	Un carattere di tipo non-word ([^w])

Ciò che è indicato in queste tabelle è soltanto un campione rappresentativo delle possibilità offerte dalle espressioni regolari: vi raccomandiamo di inserire tra i preferiti del vostro browser la pagina della documentazione JDK per **java.util.regex.Pattern**, in modo da accedere facilmente a tutti i possibili modelli.

Operatori logici

XY	X seguito da Y
X Y	X o Y
(X)	Un “gruppo di cattura” o <i>capturing group</i> . Nel prosieguo dell’espressione potrete fare riferimento all’ennesimo (<i>i</i>) gruppo catturato con \i.

Corrispondenze di delimitazione (Boundary matcher)

^	Inizio di una riga
\$	Fine di una riga
\b	Limite di parola
\B	Limite di non parola
\G	Fine della corrispondenza precedente

Come esempio, ciascuna delle espressioni regolari seguenti corrisponde alla sequenza di caratteri “Rudolph”.

```
//: strings/Rudolph.java
```

```
public class Rudolph {
    public static void main(String[] args) {
        for(String pattern : new String[]{ "Rudolph",
            "[rR]udolph", "[rR][aeiou][a-z]*", "R.*" })
            System.out.println("Rudolph".matches(pattern));
    }
} /* Output:
true
true
true
true
*///:
```

Naturalmente il vostro obiettivo non dovrà essere quello di generare l’espressione regolare più complessa che sia mai esistita, quanto di creare quella più semplice sufficiente per eseguire ciò che vi occorre. Dopo avere iniziato a comporre le vostre prime espressioni regolari, potrete utilizzare il vostro stesso codice come riferimento per scrivere nuove espressioni.

Quantificatori

Un quantificatore descrive il modo in cui un modello assimila il testo di input.

- Greedy** (avid). I quantificatori sono sempre di questo tipo, a meno di non essere stati alterati per produrre un comportamento diverso. Un’espressione “avid” trova quante più corrispondenze sia possibile con il modello (*pattern*) fornito. Un tipico errore è supporre che il vostro modello corrisponderà soltanto al primo gruppo possibile di caratteri, mentre in realtà, essendo “avid”, continuerà a ricercare finché non avrà trovato nella stringa la corrispondenza più grande possibile.
- Reluctant** (riluttante). Specificato mediante un punto interrogativo, questo quantificatore esegue la corrispondenza sul numero minimo di caratteri necessari per soddisfare il modello fornito; è chiamato anche *lazy* (pigro), *a corrispondenza minima (minimal matching, non greedy o ungreedy)*.
- Possessive** (possessivo). Attualmente questo quantificatore è disponibile soltanto in Java ed è molto avanzato, al punto che probabilmente non ve ne servirete da subito. Quando è applicata a una stringa, l’espressione regolare genera un certo numero di “stati” in modo da poterli ripristinare qualora la corrispondenza non venga trovata. I quantificatori possessivi

non mantengono questi stati intermedi, pertanto impediscono il ripristino e possono essere utilizzati per impedire che un'espressione vada “fuori controllo” e per eseguirla in modo più efficiente.

<i>Greedy</i>	<i>Reluctant</i>	<i>Possessive</i>	<i>Corrispondenze</i>
X?	X??	X?+	X, una o nessuna
X *	X *?	X*+	X, zero o più
X+	X+?	X++	X, una o più
X{n}	X{n}?	X{n}+	X, esattamente <i>n</i> volte
X{n,}	X{n,}?	X{n,}+	X, almeno <i>n</i> volte
X{n, m,}	X{n, m,}?	X{n, m,}+	X, almeno <i>n</i> volte ma non più di <i>m</i> volte

Tenete presente che l'espressione “X” spesso deve essere inclusa tra parentesi per funzionare nel modo corretto. Per esempio

abc+

potrebbe indicare che deve corrispondere alla sequenza “abc” una o più volte, e se viene applicata alla stringa di input “abcabcabc” restituirà effettivamente tre corrispondenze. In realtà, però, l'espressione indica di fare corrispondere “ab”, seguite da una o più occorrenze di “c”. Per fare corrispondere l'intera stringa “abc” una o più volte dovete scrivere:

(abc)+

Potreste essere tratti in inganno utilizzando le espressioni regolari, che costituiscono un vero e proprio linguaggio ortogonale, collocato al di sopra di Java.

CharSequence

L'interfaccia denominata **CharSequence** imposta la definizione generalizzata di una sequenza di caratteri estratta dalle classi **CharBuffer**, **String**, **StringBuffer** e **StringBuilder**.

```
interface CharSequence {
    charAt(int i);
    length();
    subSequence(int start, int end);
    toString();
}
```

Le suddette classi implementano questa interfaccia. Molte operazioni con le espressioni regolari accettano argomenti **CharSequence**.

Pattern e Matcher

Di norma compilerete “oggetti” di espressioni regolari anziché utilizzare le funzionalità di **String**, alquanto limitate. A questo scopo dovete importare **java.util.regex**, poi compilare un'espressione regolare servendovi del metodo **static Pattern.compile()**: questo produrrà un oggetto **Pattern** basato sull'espressione regolare fornita, sotto forma di argomento **String**. Per utilizzare **Pattern** chiamate il metodo **matcher()**, passandogli la stringa da ricercare. Il metodo **matcher()** produce un oggetto **Matcher**, che dispone di una serie di funzionalità, tutte elencate e descritte nella documentazione JDK per **java.util.regex.Matcher**. Per esempio, il metodo **replaceAll()** sostituisce tutte le corrispondenze con i propri argomenti.

Come primo esempio, la classe presentata di seguito può essere utilizzata per verificare le espressioni regolari a fronte di una stringa di input. Il primo argomento da riga di comando è la stringa di input sulla quale eseguire la corrispondenza, seguita da una o più espressioni regolari da applicare all'input: ricordate che in ambiente Unix/Linux le espressioni regolari da riga di comando devono essere “quotate”. Questo programma può essere utile per testare le espressioni che costruirete, al fine di verificare che si comportino nel modo previsto.

```
//: strings/TestRegularExpression.java
// Permette di testare facilmente le espressioni regolari.
// {Args: abcabcabcdefabc "abc+" " (abc)+" " (abc){2,}"}
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class TestRegularExpression {
    public static void main(String[] args) {
        if(args.length < 2) {
            print("Usage:\njava TestRegularExpression " +
                  "characterSequence regularExpression+");
            System.exit(0);
        }
        print("Input: " + args[0] + "\n");
        for(String arg : args) {
```

```

print("Regular expression: '" + arg + "'");
Pattern p = Pattern.compile(arg);
Matcher m = p.matcher(args[0]);
while(m.find()) {
    print("Match '" + m.group() + "' at positions " +
        m.start() + "-" + (m.end() - 1));
}
}
} /* Output:
} /* Output:
Input: "abcabcabcdefabc"
Regular expression: "abcabcabcdefabc"
Match "abcabcabcdefabc" at positions 0-14
Regular expression: "abc+"
Match "abc" at positions 0-2
Match "abc" at positions 3-5
Match "abc" at positions 6-8
Match "abc" at positions 12-14
Regular expression: "(abc)+"
Match "abcabcabc" at positions 0-8
Match "abc" at positions 12-14
Regular expression: "(abc){2,}"
Match "abcabcabc" at positions 0-8
*///:~

```

Un oggetto **Pattern** rappresenta la versione compilata di un'espressione regolare. Come avete visto nell'esempio precedente, potete utilizzare il metodo **matcher()** e la stringa di input per produrre un oggetto **Matcher** dall'oggetto **Pattern** compilato. **Pattern** dispone inoltre di un metodo statico **matches()**:

```
static boolean matches(String regex, CharSequence input)
```

Questo metodo serve per controllare se l'espressione regolare (**regex**) corrisponde all'intera **CharSequence input**; l'oggetto **Pattern** dispone inoltre di un metodo **split()** che produce un array di **String** in base al modo in cui è stata suddivisa la stringa secondo le corrispondenze di **regex**.

Un oggetto **Matcher** viene generato chiamando **Pattern.matcher()** con la stringa di input come argomento; l'oggetto **Matcher** viene quindi utilizzato per accedere ai risultati, per mezzo di alcuni metodi che valutano il successo o il fallimento dei diversi tipi di corrispondenza.

```

boolean matches()
boolean lookingAt()
boolean find()
boolean find(int start)

```

Il metodo **Matches()** ha successo se il modello corrisponde all'intera stringa di input, mentre **lookingAt()** ha successo se la stringa di input, a partire dall'inizio, corrisponde al pattern fornito.

Esercizio 10 (2) Per la frase “Java now has regular expressions” valutate se le seguenti espressioni troveranno una corrispondenza.

```

^Java
\Breg.*
n.w\s+h(a|i)s
s?
s*
s+
s{4}
s{1}.
s{0,3}

```

Esercizio 11 (2) Applicate l'espressione regolare:

```
(?i)((^aeiou)|(\s+aeiou))\w+?[aeiou]\b
```

a

“Arline ate eight apples and one orange while Anita hadn't any”

find()

Il metodo **Matcher.find()** può essere utilizzato per scoprire corrispondenze multiple del modello nella **CharSequence** a cui viene applicato.

```
//: strings/Finding.java
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class Finding {
    public static void main(String[] args) {
        Matcher m = Pattern.compile("\\w+")
            .matcher("Evening is full of the linnet's wings");
        while(m.find())
            printnb(m.group() + " ");
        print();
        int i = 0;
        while(m.find(i)) {
            printnb(m.group() + " ");
            i++;
        }
    }
} /* Output:
Evening is full of the linnet s wings
Evening vening ening ning ing ng is is s full full ull ll
l of of f the the he e linnet linnet innet nnet net et t s s
wings wings ings ngs gs s
*///:~
```

Il pattern “`\w+`” suddivide l’input in parole, mentre il metodo `find()` è una sorta di iteratore che si sposta in avanti nella stringa di input. Tuttavia, alla seconda versione di `find()` può essere fornito un argomento di tipo intero, che indica la posizione del carattere per l’inizio della ricerca: come potete vedere dall’output, questa versione reimposta la posizione di ricerca al valore dell’argomento.

Gruppi

I gruppi sono espressioni regolari messe in evidenza da parentesi, che possono essere chiamate in seguito per mezzo del relativo numero di gruppo; il gruppo 0 si riferisce all’intera espressione di corrispondenza, il gruppo 1 al primo gruppo tra parentesi e così via. Quindi, in

A(B(C))D

esistono tre gruppi: il gruppo 0 è **ABCD**, il gruppo 1 è **BC** e il gruppo 2 è **C**. L’oggetto `Matcher` offre alcuni metodi che forniscono informazioni sui gruppi:

public int groupCount() restituisce il numero di gruppi nel modello di matcher corrente; il gruppo 0 non è incluso in questo conteggio;

public String group() restituisce il gruppo 0 (l’intera corrispondenza) dall’operazione di corrispondenza precedente, `find()` per esempio;

public String group(int -i) restituisce il determinato numero di gruppo durante l’operazione di corrispondenza precedente. Se la corrispondenza ha avuto successo, ma se il gruppo specificato non corrisponde a nessuna parte della stringa di input, restituisce **null**;

public int start(int group) restituisce l’indice iniziale del gruppo trovato dall’operazione di corrispondenza precedente;

public int end(int group) restituisce l’indice dell’ultimo carattere, più uno, del gruppo trovato dall’operazione di corrispondenza precedente.

Considerate l’esempio seguente.

```
//: strings/Groups.java
import java.util.regex.*;
import static net.mindview.util.Print.*;
public class Groups {
    static public final String POEM =
        "Twas brillig, and the slithy toves\n" +
        "Did gyre and gimble in the wabe.\n" +
        "All mimsy were the borogoves,\n" +
        "And the mome raths outgrabe.\n\n" +
        "Beware the Jabberwock, my son,\n" +
        "The jaws that bite, the claws that catch.\n" +
        "Beware the Jubjub bird, and shun\n" +
        "The frumious Bandersnatch.";
    public static void main(String[] args) {
        Matcher m =
            Pattern.compile("(?m)(\\S+)\\s+(\\S+)\\s+(\\S+)$")
                .matcher(POEM);
        while(m.find()) {
            for(int j = 0; j <= m.groupCount(); j++)
```



```

        println("[ " + m.group(j) + "]");
        print();
    }
}
} /* Output:
[the slithy toves][the][slithy toves][slithy][toves]
[in the wabe.][in][the wabe.][the][wabe.]
[were the borogoves,][were][the
borogoves,][the][borogoves,]
[mome raths outgrabe.][mome][raths
outgrabe.][raths][outgrabe.]
[Jabberwock, my son,][Jabberwock,][my son,][my][son,]
[claws that catch.][claws][that catch.][that][catch.]
[bird, and shun][bird,][and shun][and][shun]
[The frumious Bandersnatch.][The][frumious
Bandersnatch.][frumious][Bandersnatch.]
*///:~

```

Il limerick è la prima parte del poema *Jabberwocky* di Lewis Carroll, tratto da *Attraverso lo specchio*. Potete vedere che il pattern dell'espressione regolare ha un certo numero di gruppi in parentesi costituiti da un numero arbitrario di caratteri diversi dagli spazi vuoti ("M"), seguiti da un numero arbitrario di spazi vuoti ("ls"). L'obiettivo del codice è catturare le ultime tre parole di ogni riga, la cui estremità è delimitata da "\$". Tuttavia il comportamento normale prevede la corrispondenza di "\$" con la fine dell'intera sequenza di input, pertanto dovete assicurarvi che l'espressione regolare controlli la presenza dei simboli di nuova riga all'interno della stringa di input. Questa verifica viene svolta dal flag "(?m)" all'inizio della sequenza: vedrete tra breve il funzionamento dei flag nei modelli.

Esercizio 12 (5) Modificate **Groups.java** per contare tutte le parole univocihe che non iniziano con una lettera maiuscola.

start() ed end()

Dopo una ricerca di corrispondenza andata a buon fine, il metodo **start()** restituisce l'indice iniziale della corrispondenza precedente, mentre **end()** restituisce l'indice dell'ultimo carattere trovato, più uno. L'invocazione di **start()** o **end()** in seguito a una ricerca di corrispondenza infruttuosa, o prima di ricercare una corrispondenza, produce un'**IllegalStateException**. Il program-



ma seguente dimostra anche il funzionamento di **matches()** e **lookingAt()**, analizzando una citazione dai discorsi del Comandante Taggart su Galaxy Quest.

```

//: strings/StartEnd.java
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class StartEnd {
    public static String input =
        "As long as there is injustice, whenever a\n" +
        "Targathian baby cries out, wherever a distress\n" +
        "signal sounds among the stars ... We'll be there.\n" +
        "This fine ship, and this fine crew ...\n" +
        "Never give up! Never surrender!";
    private static class Display {
        private boolean regexPrinted = false;
        private String regex;
        Display(String regex) { this.regex = regex; }
        void display(String message) {
            if(!regexPrinted) {
                print(regex);
                regexPrinted = true;
            }
            print(message);
        }
    }
    static void examine(String s, String regex) {
        Display d = new Display(regex);

```

```

Pattern p = Pattern.compile(regex);
Matcher m = p.matcher(s);
while(m.find())
    d.display("find() '" + m.group() +
        "' start = " + m.start() + " end = " + m.end());
if(m.lookingAt()) // Non e' necessario il reset()
    d.display("lookingAt() start = "
        + m.start() + " end = " + m.end());
if(m.matches()) // Non e' necessario il reset()
    d.display("matches() start = "
        + m.start() + " end = " + m.end());
}
public static void main(String[] args) {
    for(String in : input.split("\n"))
        print("input : " + in);
    for(String regex : new String[]{"\w*ere\w*",
        "\w*ever", "T\w+", "Never.*?!"})
        examine(in, regex);
}
} /* Output:
input : As long as there is injustice, whenever a
\w*ere\w*
find() 'there' start = 11 end = 16
\w*ever
find() 'whenever' start = 31 end = 39
input : Targathian baby cries out, wherever a distress
\w*ere\w*
find() 'wherever' start = 27 end = 35
\w*ever
find() 'wherever' start = 27 end = 35
T\w+
find() 'Targathian' start = 0 end = 10
lookingAt() start = 0 end = 10
input : signal sounds among the stars ... We'll be there.
\w*ere\w*

```

```

find() 'there' start = 43 end = 48
input : This fine ship, and this fine crew ...
T\w+
find() 'This' start = 0 end = 4
lookingAt() start = 0 end = 4
input : Never give up! Never surrender!
\w*ever
find() 'Never' start = 0 end = 5
find() 'Never' start = 15 end = 20
lookingAt() start = 0 end = 5
Never.*?!
find() 'Never give up!' start = 0 end = 14
find() 'Never surrender!' start = 15 end = 31
lookingAt() start = 0 end = 14
matches() start = 0 end = 31
*///:~

```

Note che **find()** localizza l'espressione regolare in qualsiasi punto dell'input, mentre **lookingAt()** e **matches()** funzionano correttamente solo se la corrispondenza dell'espressione regolare avviene subito all'inizio dell'input. Mentre **matches()** ha successo soltanto se l'intero input corrisponde all'espressione regolare, **lookingAt()** funziona unicamente per le corrispondenze con la prima parte dell'input.

Esercizio 13 (2) Modificate **StartEnd.java** in modo che utilizzi **Groups**. **POEM** come input, ma che produca output positivi per **find()**, **lookingAt()** e **matches()**.

Flag dei modelli

Un metodo alternativo, **compile()**, accetta opzioni (*flag*) che hanno effetto sul comportamento della corrispondenza:

```
Pattern Pattern.compile(String regex, int flag)
```

dove **flag** è una delle seguenti costanti di classe **Pattern**.

<i>Flag di compile()</i>	<i>Effetto</i>
Pattern.CANON_EQ (?)	Due caratteri saranno considerati corrispondenti se, e soltanto se, corrispondono le loro scomposizioni canoniche complete. Specificando questo flag, per esempio, l'espressione "u003F" corrisponderà alla stringa "?". In modalità predefinita, la valutazione di corrispondenza non tiene in considerazione l'equivalenza canonica.
Pattern.CASE_INSENSITIVE (?)	In modalità predefinita, la corrispondenza che non tiene conto di maiuscole e minuscole dà per scontato che la valutazione debba avvenire soltanto per l'insieme di caratteri US-ASCII. Questa opzione fa sì che la corrispondenza non consideri la differenza tra lettere maiuscole e minuscole. Un tipo di corrispondenza analogo, ma valido per i caratteri Unicode, può essere abilitato specificando il flag UNICODE_CASE in abbinamento a questa opzione.
Pattern.COMMENTS (?)	Con questo flag gli spazi vuoti vengono ignorati, e i commenti che iniziano con # anch'essi ignorati fino alla fine della riga.
Pattern.DOTALL (?)	In modalità dotall , l'espressione "." corrisponde a qualsiasi carattere, incluso quello di fine riga (<i>newline</i>). In modalità predefinita, invece, "." non corrisponde ai caratteri di fine riga.
Pattern.MULTILINE (?)	In modalità multiline , le espressioni "^" e "\$" corrispondono rispettivamente all'inizio e alla fine di una riga. "^" corrisponde anche all'inizio della stringa di input, e "\$" alla fine della stringa di input. In modalità predefinita queste espressioni coincidono soltanto con l'inizio e la fine dell'intera stringa di input.

<i>Flag di compile()</i>	<i>Effetto</i>
Pattern.UNICODE_CASE (?)	La corrispondenza che non tiene conto di maiuscole e minuscole, abilitata dal flag CASE_INSENSITIVE , è compatibile con la codifica Unicode standard. In modalità predefinita, questo tipo di corrispondenza tiene conto soltanto dell'insieme di caratteri US-ASCII.
Pattern.UNIX_LINES (?)	Con questo flag, soltanto il carattere di fine riga "\n" viene riconosciuto nel comportamento di ".", "^" e "\$".

Tra queste opzioni sono particolarmente utili **Pattern.CASE_INSENSITIVE**, **Pattern.MULTILINE** e **Pattern.COMMENTS**, quest'ultima soprattutto per motivi di chiarezza e documentazione. Notate che il comportamento fornito dalla maggior parte di queste opzioni si può ottenere anche inserendo all'interno dell'espressione regolare, nella posizione precedente il punto in cui volete attivarlo, i caratteri indicati tra parentesi nella tabella, per esempio (?)d). Potete combinare l'effetto di queste e di altre opzioni per mezzo dell'operatore "OR" ("|").

```
//: strings/ReFlags.java
import java.util.regex.*;

public class ReFlags {
    public static void main(String[] args) {
        Pattern p = Pattern.compile("^java",
            Pattern.CASE_INSENSITIVE | Pattern.MULTILINE);
        Matcher m = p.matcher(
            "java has regex\nJava has regex\n" +
            "JAVA has pretty good regular expressions\n" +
            "Regular expressions are in Java");

        while(m.find())
            System.out.println(m.group());
    }
} /* Output:
java
```



```
Java
JAVA
*//:/~
```

In questo esempio si crea una struttura che trova una corrispondenza con righe che iniziano con “java”, “Java”, “JAVA” ecc., e che cerca la corrispondenza di ogni riga all’interno di un insieme di righe multiple: tali corrispondenze iniziano all’inizio della sequenza di caratteri e proseguono oltre i caratteri di fine riga presenti all’interno della sequenza. Notate che il metodo **group()** restituisce soltanto la porzione di stringa corrispondente.

split()

Il metodo **split()** divide una stringa di input in un array di oggetti **String**, delimitato dall’espressione regolare.

```
String[] split(CharSequence input)
String[] split(CharSequence input, int limit)
```

Questo è un modo pratico per suddividere il testo in base a un delimitatore comune.

```
//: strings/SplitDemo.java
import java.util.regex.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class SplitDemo {
    public static void main(String[] args) {
        String input =
            "This!!unusual use!!of exclamation!!points";
        print(toString(
            Pattern.compile("!!").split(input)));
        // Elabora solo le prime tre corrispondenze:
        print(toString(
            Pattern.compile("!!").split(input, 3)));
    }
} /* Output:
```



```
[This, unusual use, of exclamation, points]
[This, unusual use, of exclamation!!points]
*//:/~
```

La seconda forma di **split()** limita il numero di suddivisioni eseguite.

Esercizio 14 (1) Riscrivete **SplitDemo** utilizzando il metodo **String.split()**.

Operazioni di sostituzione

Le espressioni regolari si rivelano particolarmente utili nella sostituzione di porzioni di testo. I metodi disponibili sono i seguenti:

replaceFirst(String replacement) sostituisce la prima parte di corrispondenza della stringa di input con **replacement**;

replaceAll(String replacement) sostituisce l’intera corrispondenza della stringa di input con **replacement**;

appendReplacement(StringBuffer sbuf, String replacement) esegue sostituzioni passo passo, in **sbuf**, anziché sostituire soltanto la prima o tutte le corrispondenze come fanno rispettivamente **replaceFirst()** e **replaceAll()**. Questo è un metodo molto importante poiché permette di chiamare metodi ed eseguire altre operazioni di elaborazione per effettuare la sostituzione.

Tenete presente che **replaceFirst()** e **replaceAll()** possono soltanto intervenire sulle stringhe fisse. Grazie a questo metodo potete selezionare i gruppi da programma e implementare sostituzioni molto potenti.

appendTail(StringBuffer sbuf, String replacement) viene invocato dopo una o più chiamate al metodo **appendReplacement()**, per copiare il resto della stringa di input.

Considerate l’esempio seguente che mostra l’impiego di tutte le operazioni di sostituzione. Il blocco di testo commentato iniziale viene estratto ed elaborato tramite le espressioni regolari per essere utilizzato come input nel resto dell’esempio.

```
//: strings/TheReplacements.java
import java.util.regex.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

/*! /*! Here's a block of text to use as input to
```

the regular expression matcher. Note that we'll first extract the block of text by looking for the special delimiters, then process the extracted block. !*/ !*/

```
public class TheReplacements {
    public static void main(String[] args) throws Exception {
        String s = TextFile.read("TheReplacements.java");
        // Esegue la corrispondenza sul blocco di commento
        // iniziale:
        Matcher mInput =
            Pattern.compile("//\\*!(.*!)\\*/", Pattern.DOTALL)
                .matcher(s);
        if(mInput.find())
            s = mInput.group(1); // Catturato da parentesi
        // Sostituisce due o piu' spazi con un solo spazio:
        s = s.replaceAll(" {2,}", " ");
        // Elimina uno o piu' spazi all'inizio di ogni riga.
        // La modalita' MULTILINE deve essere abilitata:
        s = s.replaceAll("(?m)^ ", "");
        print(s);
        s = s.replaceFirst("[aeiou]", "(VOWEL1)");
        StringBuffer sbuf = new StringBuffer();
        Pattern p = Pattern.compile("[aeiou]");
        Matcher m = p.matcher(s);
        // Elabora le informazioni di find() via
        // via che si eseguono le sostituzioni:
        while(m.find())
            m.appendReplacement(sbuf, m.group().toUpperCase());
        // Inserisce il testo rimasto:
        m.appendTail(sbuf);
        print(sbuf);
    }
} /* Output:
Here's a block of text to use as input to
the regular expression matcher. Note that we'll
first extract the block of text by looking for
```

the special delimiters, then process the extracted block.

H(VOWEL1)rE's A b10ck Of tExt t0 UsE As InpUt t0
thE rEgUlar ExprEssIOn mAtchEr. N0tE thAt wE'll
fIrst ExtrAct thE b10ck Of tExt by 100kIng fOr
thE spEcIAL dElImItErS, thEn pr0cess thE
ExtrActEd b10ck.

*/~

Il file viene aperto e letto servendosi della classe **TextFile** della libreria **net.mindview.util**, che vedrete in dettaglio nel Volume 2, Capitolo 6. Il metodo **static read()** legge l'intero file e lo restituisce come **String**; poi viene creato **mInput** per eseguire la corrispondenza di tutto il testo compreso tra “**/*!**” e “**!*/**”: notate le parentesi di raggruppamento. Quindi, le sequenze di più di due spazi vengono ridotte a uno solo e tutti gli spazi all'inizio di ogni riga vengono rimossi: per fare questo la modalità **multiline** deve essere abilitata su tutte le righe, non solo all'inizio dell'input. Le due sostituzioni vengono eseguite con l'equivalente (ma più pratico, in questo caso) **replaceAll()**, che è parte di **String**. Notate che dal momento che ogni sostituzione è utilizzata soltanto una volta nel programma, non vi è alcun onere aggiuntivo nell'adottare questa tecnica in luogo del **Pattern** precompilato.

Il metodo **replaceFirst()** esegue soltanto la prima sostituzione trovata. Tenete presente che le stringhe di sostituzione in **replaceFirst()** e **replaceAll()** sono semplici costanti letterali, che non è possibile elaborare in fase di sostituzione: se avete questa necessità dovete servirvi di **appendReplacement()**, che consente invece di integrare nel codice di sostituzione tutte le operazioni che ritenevi opportune. Nell'esempio precedente viene selezionato ed elaborato un **group()**, nel caso specifico impostando in maiuscolo le vocali trovate dall'espressione regolare, durante la costruzione del buffer **sbuff**. Generalmente eseguirete tutte le sostituzioni passo passo per poi chiamare il metodo **appendTail()**; tuttavia, se desiderate simulare il comportamento di **replaceFirst()** (o “replace *n*”), dovete effettuare una sola sostituzione e chiamare poi **appendTail()** per porre in **sbuff** il resto della stringa.

Il metodo **appendReplacement()** consente inoltre di fare riferimento ai gruppi catturati direttamente nella stringa di sostituzione, specificando “**\$g**”, in cui “**g**” è il numero del gruppo. Questa opzione, tuttavia, è consigliata per tipi di elaborazione più semplici e nel programma precedente non produrrebbe i risultati che vi attendereste.



reset()

Un oggetto **Matcher** esistente può essere applicato a una nuova sequenza di caratteri tramite i metodi di **reset()**.

```
//: strings/Resetting.java
import java.util.regex.*;

public class Resetting {
    public static void main(String[] args) throws Exception {
        Matcher m = Pattern.compile("[frb][aiu][gx]")
            .matcher("fix the rug with bags");
        while(m.find())
            System.out.print(m.group() + " ");
        System.out.println();
        m.reset("fix the rig with rags");
        while(m.find())
            System.out.print(m.group() + " ");
    }
} /* Output:
fix rug bag
fix rig rag
*///:~
```

Il metodo **reset()** senza argomenti imposta il **Matcher** all'inizio della sequenza corrente.

Espressioni regolari e Java I/O

La maggior parte degli esempi analizzati fino a questo punto ha mostrato l'utilizzo delle espressioni regolari applicate alle stringhe statiche. L'esempio seguente, invece, illustra un modo per applicare le espressioni regolari alla ricerca di corrispondenze in un file. Ispirato all'utility **grep** di UNIX, **JGrep.java** accetta due argomenti: un nome di file e l'espressione regolare di cui cercare la corrispondenza.

L'output evidenzia le righe trovate e la posizioni di corrispondenza all'interno delle singole righe.

```
//: strings/JGrep.java
// Una versione molto semplice del programma "grep".
```



```
// {Args: JGrep.java "\b[Ssct]\w+"}
import java.util.regex.*;
import net.mindview.util.*;

public class JGrep {
    public static void main(String[] args) throws Exception {
        if(args.length < 2) {
            System.out.println("Usage: java JGrep file regex");
            System.exit(0);
        }
        Pattern p = Pattern.compile(args[1]);
        // Itera le righe del file di input:
        int index = 0;
        Matcher m = p.matcher("");
        for(String line : new TextFile(args[0])) {
            m.reset(line);
            while(m.find())
                System.out.println(index++ + ":" + 
                    m.group() + ":" + m.start());
        }
    }
} /* Output: (Esempio)
0: strings: 4
1: semplice: 22
2: Ssct: 26
3: class: 7
4: static: 9
5: String: 26
6: throws: 41
7: System: 6
8: System: 6
9: compile: 24
10: String: 8
11: System: 8
12: start: 31
13: Sample: 14
*///:~
```



Il file viene aperto come oggetto **net.mindview.util.TextFile** (si veda il Volume 2, Capitolo 6) e le righe del file lette e importate in un **ArrayList**: questo significa che la sintassi `foreach` itera le righe dell'oggetto **TextFile**.

Malgrado sia possibile generare un nuovo oggetto **Matcher** all'interno del ciclo **for**, è preferibile creare un oggetto **Matcher** vuoto fuori dal ciclo, e utilizzare il metodo **reset()** per assegnare ogni riga dell'input al **Matcher**. Il risultato viene poi esplorato con **find()**.

Gli argomenti predefiniti del programma aprono il file **JGrep.java** per la lettura in input e la ricerca delle parole che iniziano con **[Ssct]**.

Uno dei migliori testi sulle espressioni regolari è *Mastering Regular Expressions*, Seconda edizione, di Jeffrey E.F. Friedl (O'Reilly, 2002). Anche su Internet sono disponibili numerose introduzioni sull'argomento, e molte informazioni utili nella documentazione dei linguaggi Perl e Python; infine, un sito di riferimento sull'argomento è RegExLib.com (www.regexlib.com), ricco di esempi.

Esercizio 15 (5) Modificate **JGrep.java** per fare in modo che accetti i flag come argomento, per esempio **Pattern.CASE_INSENSITIVE** o **Pattern.MULTILINE**.

Esercizio 16 (5) Modificate **JGrep.java** per fare in modo che accetti un nome di directory o di file come argomento: nel caso venga fornito il nome di una directory, la ricerca dovrà includere tutti i file che vi sono contenuti. Potete generare un elenco di nomi di file con:

```
File[] files = new File(".").listFiles();
```

Esercizio 17 (8) Scrivete un programma che legga il file di codice sorgente Java specificato nella riga di comando e visualizzi tutti i commenti contenuti nel codice.

Esercizio 18 (8) Scrivete un programma che legga il file di codice sorgente Java specificato nella riga di comando e visualizzi tutti i letterali stringa contenuti nel codice.

Esercizio 19 (8) Basandovi sui due esercizi precedenti, scrivete un programma che analizzi il codice sorgente Java e produca l'elenco di tutti i nomi di classe utilizzati in un determinato codice.



Scansione dell'input

Finora è stato piuttosto difficile leggere i dati da un file di testo o da standard input. La soluzione più pratica consiste nel leggere una riga di testo, scomporla in parti e utilizzare i vari metodi delle classi **Integer**, **Double** ecc. per analizzare i dati.

```
//: strings/SimpleRead.java
import java.io.*;
public class SimpleRead {
    public static BufferedReader input = new BufferedReader(
        new StringReader("Sir Robin of Camelot\n22 1.61803"));
    public static void main(String[] args) {
        try {
            System.out.println("What is your name?");
            String name = input.readLine();
            System.out.println(name);
            System.out.println(
                "How old are you? What is your favorite double?");
            System.out.println("(input: <age> <double>)");
            String numbers = input.readLine();
            System.out.println(numbers);
            String[] numArray = numbers.split(" ");
            int age = Integer.parseInt(numArray[0]);
            double favorite = Double.parseDouble(numArray[1]);
            System.out.format("Hi %.1s.\n", name);
            System.out.format("In 5 years you will be %d.\n",
                age + 5);
            System.out.format("My favorite double is %.2f.\n",
                favorite / 2);
        } catch(IOException e) {
            System.err.println("I/O exception");
        }
    }
} /* Output:
What is your name?
Sir Robin of Camelot
```



```
How old are you? What is your favorite double?
(input: <age> <double>)
22 1.61803
Hi Sir Robin of Camelot.
In 5 years you will be 27.
My favorite double is 0.809015.
*///:~
```

Il campo **input** utilizza le classi di **java.io**, che vedrete soltanto nel Volume 2, Capitolo 6; uno **StringReader** trasforma una **String** in un flusso leggibile, poi questo oggetto viene utilizzato per creare un **BufferedReader** dal momento che quest'ultimo dispone di un metodo **readLine()**. Il risultato è che l'oggetto **input** può essere letto una riga alla volta, come se si trattasse di standard input proveniente dalla console.

Il metodo **readLine()** viene utilizzato per ottenere la **String** di ogni riga di input; l'utilizzo di questo metodo è ragionevolmente semplice quando si tratta di ottenere un solo input da ogni riga di dati, tuttavia se sulla stessa riga di input sono presenti due valori le cose si complicano, e la riga stessa deve essere suddivisa in modo da analizzare separatamente ogni input: in questo caso la suddivisione ha luogo al momento della creazione di **numArray**. Tenete presente, però, che il metodo **split()** è stato introdotto con J2SE1.4: prima di questa versione di Java era necessario ricorrere ad altre tecniche.

La classe **Scanner**, aggiunta in Java SE5, elimina gran parte delle difficoltà legate alla scansione dell'input.

```
//: strings/BetterRead.java
import java.util.*;

public class BetterRead {
    public static void main(String[] args) {
        Scanner stdin = new Scanner(new StringReader("Sir Robin of
Camelot\n22 1,61803"));
        System.out.println("What is your name?");
        String name = stdin.nextLine();
        System.out.println(name);
        System.out.println(
            "How old are you? What is your favorite double?");
    }
}
```



```
System.out.println("(input: <age> <double>)");
int age = stdin.nextInt();
double favorite = stdin.nextDouble();
System.out.println(age);
System.out.println(favorite);
System.out.format("Hi %s.\n", name);
System.out.format("In 5 years you will be %d.\n",
    age + 5);
System.out.format("My favorite double is %.2f.",
    favorite / 2);
}
} /* Output:
What is your name?
Sir Robin of Camelot
How oald are you? What is your favorite double?
(input: <age> <double>)
22
1.61803
Hi Sir Robin of Camelot.
In 5 years you will be 27.
My favorite double is 0.809015.
*///:~
```

Il costruttore di **Scanner** accetta praticamente qualunque genere di oggetto di input, inclusi un oggetto **File** (si veda il Volume 2, Capitolo 6), un **InputStream**, una **String** o in questo caso un **Readable**; quest'ultimo è un'interfaccia introdotta in Java SE5 per descrivere “qualcosa che abbia un metodo **read()**”: il **BufferedReader** dell'esempio precedente rientra appunto in questa categoria.

Con **Scanner**, fasi di input, scomposizione in porzioni e scansione sono tutte nascoste in diversi tipi di metodi “next”. Un normale metodo **next()** restituisce la successiva porzione di **String** e altri metodi “next” sono disponibili per tutti i tipi primitivi, tranne **char**, nonché per **BigDecimal** e **BigInteger**.

Tutti i metodi “next” sono bloccanti, intendendo con questa espressione che ritorneranno (**return**) soltanto dopo che una porzione di dati completa sia disponibile per l'input. Esistono anche corrispondenti metodi “has-



Next" che restituiscono **true** se la porzione di input successiva è del tipo corretto.

Una differenza interessante tra i due esempi precedenti è la mancanza di un blocco **try** per **IOExceptions** in **BetterRead.java**. Una delle supposizioni fatte da **Scanner** è che un'**IOException** segnala la fine dell'input, pertanto queste eccezioni vengono "inghiottite" dall'oggetto **Scanner**. In ogni caso l'eccezione più recente è disponibile tramite il metodo **ioException()** e può essere eventualmente esaminata.

Esercizio 20 (2) Create una classe che contenga campi di tipo **int**, **long**, **float** e **double** e alcune **String**. Generate un costruttore per questa classe che accetti un solo argomento **String** e analizzate questa stringa, suddividendola nei vari campi che la compongono. Infine aggiungete un metodo **toString()** e dimostrate che la vostra classe funziona correttamente.

Separatori di Scanner

In modalità predefinita, una classe **Scanner** suddivide l'input in base agli spazi vuoti, ma potete anche specificare il vostro delimitatore (separatore) sotto forma di espressione regolare.

```
//: strings/ScannerDelimiter.java
import java.util.*;

public class ScannerDelimiter {
    public static void main(String[] args) {
        Scanner scanner = new Scanner("12, 42, 78, 99, 42");
        scanner.useDelimiter("\\s*,\\s*");
        while(scanner.hasNextInt())
            System.out.println(scanner.nextInt());
    }
} /* Output:
12
42
78
99
42
*//*//:~
```



Come separatore durante la lettura della **String** questo esempio utilizza le virgolette, circondate da un numero arbitrario di spazi vuoti; la stessa tecnica può essere adottata per leggere dai file separati da virgolette. Oltre al metodo **useDelimiter()** per l'impostazione del modello di separatore è disponibile anche **delimiter()**, che restituisce il **Pattern** utilizzato come separatore corrente.

Scansione con le espressioni regolari

In aggiunta alla scansione basata sui tipi primitivi predefiniti, Java consente di ricercare la corrispondenza su pattern personalizzati, che si rivelano particolarmente utili quando occorra analizzare dati più complessi. L'esempio mostrato di seguito analizza i dati di log prodotti da un firewall.

```
//: strings/ThreatAnalyzer.java
import java.util.regex.*;
import java.util.*;

public class ThreatAnalyzer {
    static String threatData =
        "58.27.82.161@02/10/2005\n" +
        "204.45.234.40@02/11/2005\n" +
        "58.27.82.161@02/11/2005\n" +
        "58.27.82.161@02/12/2005\n" +
        "58.27.82.161@02/12/2005\n" +
        "[Next log section with different data format]";
    public static void main(String[] args) {
        Scanner scanner = new Scanner(threatData);
        String pattern = "(\\d+[.]\\d+[.]\\d+[.]\\d+)@"
            + "(\\d{2}/\\d{2}/\\d{4})";
        while(scanner.hasNext(pattern)) {
            scanner.next(pattern);
            MatchResult match = scanner.match();
            String ip = match.group(1);
            String date = match.group(2);
            System.out.format("Threat on %s from %s\n", date, ip);
        }
    }
} /* Output:
```



```
Threat on 02/10/2005 from 58.27.82.161
Threat on 02/11/2005 from 204.45.234.40
Threat on 02/11/2005 from 58.27.82.161
Threat on 02/12/2005 from 58.27.82.161
Threat on 02/12/2005 from 58.27.82.161
*///:~
```

Quando utilizzate **next()** con un modello specifico, ne viene cercata la corrispondenza a fronte del successivo token di input: il risultato viene poi reso disponibile con il metodo **match()** e, come potete vedere nell'esempio, funziona esattamente come l'espressione regolare esaminata in precedenza.

Tuttavia la scansione mediante espressioni regolari presenta un inconveniente, in quanto il modello viene fatto corrispondere soltanto con il token di input seguente: pertanto, se il vostro pattern contenesse un separatore la corrispondenza non verrebbe mai rilevata.

StringTokenizer

Prima dell'introduzione delle espressioni regolari, in J2SE1.4, e della classe **Scanner**, in Java SE5, la tecnica utilizzata per suddividere una stringa prevedeva la sua scomposizione in porzioni mediante la classe **StringTokenizer**.

Ora, come sapete, la stessa operazione può avvenire in modo molto più facile e rapido ricorrendo alle espressioni regolari o alla classe **Scanner**. Di seguito è mostrato un semplice confronto tra **StringTokenizer** e le altre due tecniche.

```
//: strings/ReplacingStringTokenizer.java
import java.util.*;

public class ReplacingStringTokenizer {
    public static void main(String[] args) {
        Stringa input = "But I'm not dead yet! I feel happy!";
        StringTokenizer stoke = new StringTokenizer(input);
        while(stoke.hasMoreElements())
            System.out.print(stoke.nextToken() + " ");
        System.out.println();
    }
}
```



```
System.out.println(Arrays.toString(input.split(" ")));
Scanner scanner = new Scanner(input);
while(scanner.hasNext())
    System.out.print(scanner.next() + " ");
}
} /* Output:
But I'm not dead yet! I feel happy!
[But, I'm, not, dead, yet!, I, feel, happy!]
But I'm not dead yet! I feel happy!
*///:~
```

Le espressioni regolari e gli oggetti **Scanner** consentono anche di scomporre una stringa in parti utilizzando pattern molto più complessi, un risultato molto difficile da ottenere per mezzo di **StringTokenizer**: si può quasi affermare, quindi, che l'impiego di **StringTokenizer** sia ormai obsoleto.

Riepilogo

In passato il supporto offerto da Java alla manipolazione delle stringhe era alquanto rudimentale, tuttavia nelle versioni più recenti del linguaggio è decisamente migliorato, integrando soluzioni caratteristiche di altri linguaggi: oggi le funzionalità per la gestione delle stringhe sono piuttosto complete, anche se talvolta occorre riservare la massima attenzione alle prestazioni, in particolare avendo cura di utilizzare **StringBuilder** in modo corretto.

*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



Il volume *Java - Ora e Poi* è composto da esempi di codice che illustrano i concetti di programmazione Java. Tuttavia, non tutti gli esempi sono disponibili nel volume. Alcuni esempi sono stati rimossi per ragioni di spazio o di sicurezza. Inoltre, alcuni esempi sono stati modificati per adattarli alla versione specifica di Java utilizzata nel volume.

Per questo motivo, è stato deciso di fornire supplementi aggiuntivi per i lettori che desiderano approfondire i concetti visti nel volume. I supplementi includono:

- Documentazione: una guida dettagliata alle classi e ai metodi presenti nel volume.
- Seminari: una serie di video e di documenti che spiegano i concetti visti nel volume.
- Servizi: un sito web dove i lettori possono postare domande e ricevere risposte da esperti.

Questo appendice presenta i supplementi disponibili, consentendo ai lettori di valutare se essi saranno utili per loro.

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*);



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

abstract, classe 326
Adapter design pattern 351
Adapter Method idiom 466-470
add(), metodo 416
Adventure, classe 345
algoritmo di hashing 445
aliasing 80-81
ambito di visibilità 48
AND 89
APL 2
array 23, 47, 426
 associativi 23
 visualizzazione e stampa di 426-428
ArrayList, classe 211
Assembler 2
astrazione 2-4
autoboxing 46
avanzamento carta (*form feed*), carattere 503

B

BASIC 2
binding 18, 290-291
 anticipato 18, 290
 dinamico 18

ritardato 290
statico 18
break, parola chiave 134-135
browser di classi (*class browser*) 232

C

C 2
callback 394-397
carattere di nuova riga (*newline*) 132
caratteri di sottolineatura (*underscore*) 71
casting 20, 106
CGI (*Common Gateway Interface*) 32
CharSequence, interfaccia 506
classi 4, 11, 232, 239, 325, 363
 accesso alle 232-237
 accesso protected 264-265
 a nidificazione multipla, accesso alle 390
 astratte 325-330
 combinazione di composizione ed ereditarietà nelle 252-255
delega 250-252
di base 11
esterne 363
figlie 11
garantire un cleanup



corretto nelle 255-259
inizializzazione
con ereditarietà 280-282
inizializzazione e
caricamento 279-282
interne 363-414
anonime 377-386
collegamento alla classe
esterna delle 366-369
creazione delle 364-366
e framework
di controllo 397-406
ereditare dalle 406-407
e upcasting 371-374
identificatori delle 412
locali 409-412
metodi e ambiti 374-377
motivi dell'utilizzo
delle 391-394
sovrascrittura delle 407-409
nidificate 386-390
all'interno
di interfacce 388-389
occultamento del nome 259
scelta tra composizione
ed ereditarietà 261-263
sintassi dell'ereditarietà 244-246
sintassi
della composizione 240-244
superclasse,
inizializzazione della
247-250
superiori/genitore 11
upcasting 265-266
CLASSPATH 215, 217, 218
Cleanser, classe 245
cleanup 147-208
finalizzazione e garbage
collection 168-177
closure 394-397

coda con priorità
(*priority queue*) 455
coda doppia (*double-ended queue*,
o deque) 439
codice 212
organizzazione del 212-214
Collection
confronto di,
con Iterator 458-463
collezione 423
aggiunta di gruppi
di elementi a 423-425
commenti 63-70
condizione di terminazione
171-173
contenere gli oggetti 415-474
contenitori 22-25
contenitori type-safe 416-421
continue, parola chiave 134-135
controllo di accesso 209-238
conversione
con ampliamento (*widening conversion*) 107
con riduzione (*narrowing conversion*) 107
Cookie, classe 229, 230
Cooling System, classe 16
costruttori 160, 303
chiamate di,
da costruttori 165-167
comportamento dei metodi
polimorfi nei 312-315
e polimorfismo 303-315
predefiniti 160-162

D

dati 50
creazione di tipi di 50-52

Depth, classe 180
design pattern 235, 466
disciplina di accodamento
(*queuing discipline*) 421, 455
dispose(), metodo 306, 310
dissociazione completa 335-342
do-while 126-127
DotNew, classe 370
downcasting 24, 321-323
e tipi di informazioni
a runtime 321-323

E

eccezioni 28-29
gestione delle 28-29
elaborazione transazionale 31
ereditarietà 11-15, 239, 306
confronto tra sostituzione
ed estensione 318-320
e cleanup 306-312
multipla 342-344, 343
progettazione in funzione della
316-318
sintassi della 244-246
errori 28-29
trattamento degli 28-29
esecuzione 123-146
espressione booleana 124
espressioni regolari 498-524
creazione di 503-505
e Java I/O 522-524
flag dei modelli 515-517
gruppi 511-516
Matcher 507-517
Pattern 507-517
per la scansione dell'input 529-530
per operazioni
di sostituzione 519-522

principi fondamentali delle 499-503
quantificatori nelle 505-506
reset() 522-523
split(), metodo 518-519
estensibilità
dei programmi 296-299

F

Factory Method 383-386
Factory Method design pattern 358
false 123-124
FIFO (*First-In, First-Out*) 453
final 268-279
argomenti 273-274
campi, non inizializzati 271-273
classi di tipo 277-278
considerazioni su 278-279
dati di tipo 268-271
e private 275-277
metodi di tipo 274
parola chiave 268
finalize() 169
funzione di 169-170
firma (*signature*) 53
flag dei modelli 515-517
for 127-129
Foreach e iteratori 462-469
Foreign, classe 237
FORTRAN 2, 95

G

garbage collection 61, 173
funzionamento della 173-177
garbage collector 22, 27, 49, 147
generici 416-420. Vedere tipi
parametrizzati



gerarchia a radice comune 21
GIF (*Graphics Interchange Format*) 33
goto, parola chiave 136-140
GUI (*Graphical User Interface*) 35, 398

H

HTML (*HyperText Markup Language*) 32

I

if-else 124-125
implementazione 8-11, 231
e interfaccia 231-232
nascosta 8-10
riutilizzo 10-11
importazione statica
(*static import*) 76
InitialValues, classe 179
inizializzazione 147-208
dei membri 177-181
del costruttore 181-182
di array 191-196
di dati static 183-186
di istanze non static 189-190
garantita dal costruttore 148-150
ordine di 182-183
per la creazione di elenchi di argomenti variabili 197-204
specificata 179-181
static esplicita 187-188
input, scansione del 525-530
con le espressioni regolari 529-530
con separatori di Scanner 528
Instrument, classe 265, 325, 327

interfacce 325-362
adattarsi alle 349-353
campi nelle 352-354
inizializzazione dei 353
collisione di nomi nella combinazione di 347-348
e factory 358-362
estensione di, mediante ereditarietà 345-346
nidificate 354-357
interface, parola chiave 331
intranet 38
istruzione di selezione 141
Iterator 434-438
confronto di,
con Collection 458-463
ListIterator, sottotipo di 437-438
iteratore inverso (*reverse iterator*) 466
iterazione 125-129

J

JAR (*Java ARchive*) 213
Java 30
e Internet 30-39
java.lang, classe 128
Javadoc 64
JDK (*Java Developer's Kit*) 62
JIT (*Just-In-Time*) 176
JRE (*Java Runtime Environment*) 36
JSP (*Java Server Pages*) 39
JVM (*Java Virtual Machine*) 37

L

lazy initialization (inizializzazione pigrta) 242

libreria di contenitori 421
Collection(collezione) 421
Map(mappa) 421
LIFO (*Last-In, First-Out*) 441
lightweight object (oggetto leggero) 434

linguaggi di programmazione 2
APL 2
LISP 2
linguaggi di scripting 34-35
LinkedList 439-441
LISP 2
List 429-434
ListIterator 437-438
lowerCamelCase 71

M

macchina virtuale Java (*Java Virtual Machine*) 136
Map 448-453
mapping 448
membro dati 7, 50
meno unario 84
metodi 53, 325
argomenti 53-55
astratti 325-330
overloaded 153-154
valori di ritorno 53-55
MI (*Multiple Inheritance*) 413
modellizzazione 2
modificatori di accesso 223-231
package access 223-224
package predefinito 225-226
private 226-228
protected 228-231
public 224-225
MyClass, classe 213

N

.new, utilizzo di 369-371
non-word, carattere 501
NOT 89

O

Object, classe 27
occultamento dell'implementazione (*implementation hiding*) 231
oggetti 1-40, 3, 5, 42
ambito di visibilità 49
creazione 43-47
creazione e durata degli 25-27
distruzione dei 47-48
intercambiabilità degli 17-21
interfaccia 4-7
manipolazione per riferimento 42-43

persistenti 44
servizi forniti da 7-8
streamed 44
String 42, 43
test sull'equivalenza degli 87-89

OOP (*Object-Oriented Programming*) 1-2
operatore virgola 129
operatori 75-122
assegnazione 78-81
bit a bit 96-97
di cast 106-109
promozione 109
troncamento e arrotondamento 107-109
di complemento a uno 96
di shift sinistro (<<) 97
di spostamento o di scorrimento (*shift operator*) 97

di String + e += 104-105
 di tipo shift 97-102
 logici 89-92
 cortocircuito 91-92
 matematici 81-85
 per l'incremento e decremento
 automatici 85-86
 precedenza 77-78
 problemi ricorrenti
 nell'utilizzo degli 105-106
 relazionali 86-89
 sovraccarico degli 104
 ternari: if-else 102-103
 unari di segno positivo
 e negativo 84-85
 utilizzo degli 77
 operazioni di sostituzione 519-522
 appendReplacement(StringBuffer
 sbuff, String replacement),
 metodo 519
 replaceAll(String replacement),
 metodo 519
 replaceFirst(String replacement),
 metodo 519
 OR 89
 ordinamento lessicografico 447
 output, formattazione del 487-498
 classe Formatter 489-490
 conversioni
 di Formatter 492-496
 modificatori di formato 490-492
 printf() 488
 String.format() 496-498
 System.out.format() 488-489
 overflow 312
 overloading 132, 150
 con primitivi 154-159
 dei metodi 150-160
 sui valori di ritorno 159-160

P
 package 211-223
 anonimo 212
 collisioni 218-219
 creazione di nomi univoci
 214-218
 librerie personalizzate
 di strumenti 219-222
 osservazioni sui 222-223
 parametrato (*parameterized type*) 442
 perdite di memoria (*memory leaks*) 27
 più unario 84
 plug-in 34
 polimorfismo 17-21, 285-324
 e costruttori 303-315
 print 75-77
 Print, classe 219
 PriorityQueue 455-458
 private, metodi 300
 sovrascriftura dei 300
 Processor, classe 336
 profiler 132, 432
 programma Java 55
 costruzione di 55-63
 programmatore client 8
 programmazione 29
 concorrente 29-30
 lato client 32-33
 lato server 39
 programmazione orientata agli oggetti (*OOP, Object-Oriented Programming*) 1
 public, classe 212
 puntatore allo stack (*stack pointer*) 44

Q

queue 453-458
 Queue, interfaccia 454

R

RAM (*Random Access Memory*) 44
 Random, classe 448
 Random, oggetto 84
 relazioni 15-17
 is-a 15-17
 is-like-a 15-17
 reset(), metodo 522-523
 return, parola chiave 133-134
 ricorrenza
 non intenzionale 483-484
 riferimento nominativo
 (*named reference*) 415
 ritorno di carrello (a capo), carattere 503
 ROM (*Read Only Memory*) 44

S

salto incondizionato (*unconditional branching*) 133
 Scanner, classe 349
 scansione dell'input 525-530
 con le espressioni
 regolari 529-530
 con separatori di Scanner 528
 schemi di suddivisione (*split pattern*) 502
 Set 444-448
 Shape, classe 295
 sintassi 65-66
 sintassi foreach 130-132

size(), metodo 417
 sizeof 109
 Soup1, classe 235
 Soup2, classe 235
 sovraccarico
 (*overloading*) 348, 477
 sovrascrittura (*overriding*) 348
 spazio della soluzione 2
 spazio del problema 2
 spazio di nomi (*namespace*) 211
 specificatori di accesso
 (*access specifier*) 210
 specificatori di formato
 (*format specifier*) 488
 split(), metodo 518-519
 stack 441-444
 di tipo pushdown 441
 Stack, classe 212, 442-444
 static 57-59
 meodi e campi 300-303
 static, significato 167-168
 StaticInitialization, classe 186
 stile di codifica 71-72
 STL (*Standard Template Library*) 23
 Strategy design pattern 349
 String, classe 77, 131, 475, 477,
 484, 500
 String, oggetto 61, 240
 StringBuilder 477
 confronto tra sovraccarico
 di "+" e 477-483
 stringhe 475-532
 invariabili 475-476
 operazioni sulle 485-487
 charAt() 485
 compareTo() 485
 concat() 486
 contains() 485
 contentEquals() 485



Costruttore 485
endsWith() 486
equals() 485
equalsIgnoreCase() 485
equalsIgnoreCase() 485
getBytes() 485
getChars() 485
indexOf() 486
intern() 487
lastIndexOf() 486
length() 485
regionMatches() 486
replace() 486
startsWith() 486
subSequence() 486
substring() 486
toCharArray() 485
toLowerCase() 487
toUpperCase() 487
trim() 487
valueOf() 487
StringTokenizer, classe 530-531
switch 141-145

T

tag 67
@author 68
@deprecated 70
@param 69
@return 69

@see 67
@since 69
@throws 69
{@docRoot} 68
Tank, classe 79
this, parola chiave 162-168
.this, utilizzo di 369-371
tipi di ritorno covarianti 315-316
tipi enumerativi 204-207
tipi parametrizzati 24-25
tipi primitivi 45-46
true 123-124

U

UML (*Unified Modeling Language*) 7
upcasting 286
revisione 286-290
UpperCamelCase 71

V

valori letterali 92-96
notazione esponenziale 94-96
variabili automatiche (scoped) 26

W

Widget, classe 232, 233