Thinking in Java

第二版

Bruce Eckel
President, MindView, Inc.

侯捷 / 王建興
合譯

讀者回應：

此書遠勝我見過的所有 Java 書。非常完整地奉行重要性順序…，搭配恰到好處的範例，以及睿智而不呆板的解說…。和其他許多 Java 書籍對照起來，我認爲它超乎尋常地成熟，前後文相互呼應，技術上誠實，文筆流暢、用語精確。以我個人觀點，我認爲它是 Java 書籍中的完美典型。**Anatoly Vorobey, Technion University, Haifa, Israel**

本書是我所見過的（含括任何語言的）程式設計入門書籍中，最好的一本。**Joakim Ziegler, FIX sysop**

謹對這樣一本精采至極的 Java 書籍，獻上我最真誠的感謝。**Dr. Gavin Pillay, Registrar, King Edward VIII Hospital, South Africa**

再次感謝您這本令人讚嘆不已的書籍。我曾經一度陷入困境（因爲我並不是一個熟悉 C 語言的程式員），但是這本書給了我很大幫助，讓我能夠盡可能調快自己的閱讀速度。對我而言，一開始就能夠理解底層的原理和觀念，真是太酷了。我再也不必經由錯誤嘗試來建立心中的觀念模型。我真的滿心期盼能夠在不久的將來，參加您的研討課程。**Randall R. Hawley, Automation Technician, Eli Lilly & Co.**

我所見過的電腦書籍中，最棒的一本。**Tom Holland**

我所讀過的眾多程式設計絕佳好書中的一本...。以 Java 爲主題的書籍中，最好的一本。**Ravindra Pai, Oracle Corporation, SUNOS product line**

Java 書籍中，我生平所見最佳的一本。您成就的作品真是太偉大了。您的深度令人讚嘆。此書一旦付梓，一定會出現在我的購買清單中。96 年 10 月起我就開始學習 Java，這段期間我讀過幾本書，您的這本肯定是「必讀」的一本。過去幾個月來，我們集中心力於一個完全以 Java 開發的產品，您這本書厚實了我過去以來未能紮穩根基的種種主題，擴展了我的知識基礎。和合作包商面試時，我甚至引用了您的內容，做爲參考的依據。向他們詢問我自這本書中讀來的資料，便可以掂出他們對 Java 的了解程度（例如，array 和 Vector 之間有何差異）。這本書真是無與倫比！**Steve Wilkinson, Senior Staff Specialist, MCI Telecommunications**

好書好書！生平所見的 Java 最佳書籍。**Jeff Sinclair, Software Engineer, Kestral Computing**

向您的《*Thinking in Java*》致上謝意。因爲有您的付出，將單純的語言描述轉化爲富有思想、具洞察力、分析透徹的入門指引，而不必向「那個公司」卑躬屈膝。我幾乎讀過所有其他書籍 – 但只有您和 Patrick Winston 的作品，能在我的心中佔有一席之地。我總是將它推薦給其他客戶。再次謝謝您。**Richard Brooks, Java Consultant, Sun Professional Services, Dallas**

其他書籍只涵蓋了 Java 的 WHAT（只探討了語法和相關類別庫），或者只包含了 Java 的 HOW（實際的程式範例）。《*Thinking in Java*》則是我所知道的書籍中，對 Java 的 WHY 做出詳盡解說的唯一一本。爲什麼要這麼設計、爲什麼它會那麼運作、爲什麼有時候會發生問題、爲什麼它在某些地方比 C++好而某些地方不會。雖然教授程式語言的 WHAT 和 HOW 也很重要，但是《*Thinking in Java*》肯定是愛好思想的人們在 Java 書籍中的唯一選擇。**Robert S. Stephenson**

這麼一本了不起的書，真是叫人感激不盡。愈讀愈叫人愛不釋手。我的學生們也很喜歡呢。**Chuck Iverson**

我只是想讚揚您在《*Thinking in Java*》上的表現。人們喜歡您看重 Internet 的未來遠景，而我只是想感謝您的付出與努力。真是感激不盡啊。**Patrick Barrell, Network Officer Mamco, QAF Mfg. Inc.**

大多數市面上的 Java 書籍，對於初學用途還算恰當。但大多數也都僅僅到達那樣的層次，而且有著許多相同的範例。您的書籍顯然是我所見過最佳的進階思考書籍。快點出版吧！由於《*Thinking in Java*》的緣故，我也買了《*Thinking in C++*》。**George Laframboise, LightWorx Technology Consulting, Inc.**

關於您的《*Thinking in C++*》（當我工作的時候，它總是在書架上佔有最顯著的位置），先前我便曾提筆告訴過您我對它的喜愛。現在，我仔細鑽研了您的電子書，我還必須說：「我愛死它了！」頗具知性與思辨，閱讀

起來不像是無味的教科書。您的書中涵蓋了 Java 開發工作中最重要、最核心的觀念：事物的本質。**Sean Brady**

您的例子不僅清楚，而且易於理解。您細膩地照顧到 Java 之中十分重要的細節，而這些細節在其他較差的 Java 文件中是無法找到的。由於您已經假設了必備的基本程式設計概念，讀者的時間也不至於被虛耗。**Kai Engert, Innovative Software, Germany**

我是《*Thinking in C++*》的忠實書迷，我也將這本書推薦給我的朋友們。當我讀完您的 Java 書籍電子版時，我真的覺得，您總是維持著高水準的寫作品質。謝謝您！**Peter R. Neuwald**

寫得超棒的 Java 書籍...我認爲您寫作此書的成就實在不凡。身爲芝加哥區域的 Java 特殊愛好者團體領導，我已經在我們最近的聚會中讚揚您的這本書和您的網站許多次了。我想將《*Thinking in Java*》做爲 SIG（譯註：Special Interest Group，特殊愛好者團體）每月聚會的基礎。在聚會中，我們將依章節順序，進行溫習與討論。**Mark Ertes**

對於您的努力成果，我由衷感激。您的書是佳作，我將這本書推薦給我們這兒的使用者與博士班學生。**Hugues Leroy // Irisa-Inria Rennes France, Head of Scientific Computing and Industrial Tranfert**

截至目前，雖然我只讀了約 40 頁的《*Thinking in Java*》，卻已經發現，這本書絕對是我所見過內容呈現方式最爲清晰的一本程式設計書籍...我自己也是一個作者，所以我理應表現出些許的挑剔。我已訂購《*Thinking in C++*》，等不及要好好剖析一番 – 對於程式設計這門學問，我還算是新手，因此事事都得學習。這不過是一篇向您的絕佳作品致謝的簡短書信。在痛苦地遍覽大多數令人食而生厭、平淡無味的電腦書籍後，雖然有些書籍有著極佳的口碑，我對於電腦書的閱讀熱情卻一度消褪。不過，現在我又再度重拾信心。**Glenn Becker, Educational Theatre Association**

謝謝您讓這麼一本精采的書籍降臨人世。當我困惑於 Java 和 C++ 的同時，這本書對於最終的認識提供了極大助益。閱讀您的書令人如沐春風。**Felix Bizaoui, Twin Oaks Industries, Louisa, Va.**

能夠寫出這麼優秀的作品，除了向您道賀，實在說不出什麼了。基於閱讀《*Thinking in C++*》的經驗，我決定讀讀《*Thinking in Java*》，而事實證明它並未讓人失望。**Jaco van der Merwe, Software Specialist, DataFusion Systems Ltd, Stellenbosch, South Africa**

本書無疑是我所見過最佳的 Java 書籍之一。**E.F. Pritchard, Senior Software Engineer, Cambridge Animation Systems Ltd., United Kingdom**

您的書籍使我曾讀過、或曾草草翻閱過的其他 Java 書籍，似乎都變得加倍的無用、該罵。 **Brett g Porter, Senior Programmer, Art & Logic**

我已經持續閱讀您這本書一兩個星期了。相較於之前我所讀過的 Java 書籍，您的書籍似乎更能夠給我一個絕佳的開始。請接受我的恭喜，能寫出這樣一本出色的作品，真不容易。**Rama Krishna Bhupathi, Software Engineer, TCSI Corporation, San Jose**

只是很想告訴您，這本書是一部多麼傑出的作品。我已將這本書做爲公司內部 Java 工作上的主要參考資料。我發現目錄的安排十分適當，讓我可以很快找出我想要的章節。能夠看到一本書籍，不是只重新改寫 API，或只是把程式員當做呆瓜，真是太棒了。**Grant Sayer, Java Components Group Leader, Ceedata Systems Pty Ltd, Australia**

噢！這是一本可讀性高、極富深度的 Java 書籍。坊間已經有太多品質低落的 Java 書籍。其中雖然也有少數不錯的，但在我看過您的大作之後，我認爲它勢必是最好的。**John Root, Web Developer, Department of Social Security, London**

我才剛開始閱讀《*Thinking in Java*》這本書。我想它肯定好到了極點，因爲我愛死了《*Thinking in C++*》（我以一個熟悉 C++、卻渴望積極提升自己能力的程式員的身份，來閱讀這本書）。我對 Java 較爲陌生，但是預料必能感到滿意。您是一位才華洋溢的作家。**Kevin K. Lewis, Technologist, ObjectSpace, Inc.**

我想這是一本了不起的書。我從這本書學到了所有關於 Java 的知識。多謝您讓大家可以從 Internet 上免費取得。如果沒有您的付出，我至今仍然對

Java 一無所知。本書最棒的一點，莫過於它同時也說明了 Java 不好的一面，而不像是一本商業宣傳。您的表現真是優秀。**Frederik Fix, Belgium**

我無時無刻不沉浸在您的書籍之中。幾年以前，當我開始學習 C++ 時，是《*C++ Inside & Out*》帶領我進入 C++ 的迷人世界裡。那本書幫助我的生命得到更好的種種機會。現在，在追尋更進一步知識的同時，當我興起了學習 Java 的念頭，我無意中又碰見了《*Thinking in Java*》— 毫無疑問，我的心中再不認爲還需要其他書籍。就是這麼的令人難以置信。持續閱讀此書的過程，就像重新發掘自我一樣。我學習 Java 至今已經一個月，現在對 Java 的體會日益加深，這一切都不得不由衷感謝您。**Anand Kumar S., Software Engineer, Computervision, India**

您的作品做爲一般性的導論，是如此卓越出眾。**Peter Robinson, University of Cambridge Computer Laboratory**

本書內容顯然是我所見過 Java 教學書籍中最頂尖的。我只是想讓您知道，我覺得能夠碰上這本書，自己有多麼幸運。謝謝！**Chuck Peterson, Product Leader, Internet Product Line, IVIS International**

了不起的一本書。自從我開始學習 Java，這已經是第三本書了。至今，我大概閱讀了三分之二，我打算好好把這本書讀完。我能夠找到這本書，是因爲這本書被用於 Lucent Technologies 的某些內部課程，而且有個朋友告訴我這本書可以在網絡上找到。很棒的作品。**Jerry Nowlin, MTS, Lucent Technologies**

在我所讀過的六本 Java 書籍中，您的《*Thinking in Java*》顯然是最好也最明白易懂的。**Michael Van Waas, Ph.D., President, TMR Associates**

關於《*Thinking in Java*》，我有說不盡的感謝。您的作品真是精采超乎尋常啊！更不必說這本書可以從網絡免費下載了！以學生的身份來看，我想您的書籍的確是無價珍寶（我也有一本《*C++ Inside & Out*》，這是一本偉大的 C++ 書籍），因爲您的書不僅教導我應該怎麼做，也教導我背後之所以如此的原因所在，而這對於 C++ 或 Java 學習者建立起堅固基礎是相當重要的。我有許多和我一樣喜愛程式設計的朋友，我也對他們提起您的

書。他們覺得真是太棒了！再次謝謝您！順道一提，我是印尼人，就住在爪哇。**Ray Frederick Djajadinata, Student at Trisakti University, Jakarta**

單是將作品免費置放於網絡的這種氣度，就使我震驚不已。我想，我應該讓您知道，對於您的作爲，我是多麼感激與尊敬。**Shane LeBouthillier, Computer Engineering student, University of Alberta, Canada**

每個月都在期待您的專欄。我必須告訴您，我有多麼盼望。做爲物件導向程式設計領域的新手，我感謝您花在種種最基礎主題上的時間和思考。我已經下載了您的這本書，但我會在這本書出版的同時，馬上搶購。對於您的種種幫助，我感謝於心。**Dan Cashmer, B. C. Ziegler & Co.**

能夠完成這麼了不起的作品，可喜可賀。首先，我沒能搞定《*Thinking in Java*》的 PDF 版本。甚至在我全部讀完之前，我還跑到書店，找出了《*Thinking in C++*》。我已從事電腦相關工作超過八年，做過顧問、軟體工程師、教師/教練，最近則當起了自由業。所以我想我的見識理應足夠（並非「看盡千帆」，而是「足夠」）。不過，這些書使得我的女朋友稱我爲「怪人」。我並不反對，只不過我認爲我已經遠超過這個階段。我發現我是如此地沉浸在這兩本書中，和其他我曾接觸過、買過的電腦書籍相比，大大的不同。這兩本書都有極佳的寫作風格，對於每個新主題都有很好的簡介與說明，而且書中充滿睿智的見解。幹得好。**Simon Goland, simonsez@smartt.com, Simon Says Consulting, Inc.**

對於您的《*Thinking in Java*》，我得說，真是太了不起了。它就是我苦尋許久而不可得的那種書。尤其那些針砭 Java 軟體設計良窳的章節，完全就是我想要的。**Dirk Duehr, Lexikon Verlag, Bertelsmann AG, Germany**

謝謝您的兩本了不起的作品：《*Thinking in C++*》和《*Thinking in Java*》。我在物件導向程式設計上的大幅進步，得自於您的助益最多。**Donald Lawson, DCL Enterprises**

多謝您願意花費時間來撰寫這麼一本大有用處的 Java 書籍。如果教學能夠讓您明白某些事情的話，那麼，此刻，您肯定對自己極爲滿意。**Dominic Turner, GEAC Support**

我曾讀過的最棒的 Java 書籍 — 我真的讀過不少。**Jean-Yves MENGANT, Chief Software Architect NAT-SYSTEM, Paris, France**

《*Thinking in Java*》涵蓋的內容與所做的解說絕對是最好的。極易閱讀，連程式碼都如此。**Ron Chan, Ph.D., Expert Choice, Inc., Pittsburgh PA**

您的書極好。我讀過許多談論程式設計的書籍，但是您的這本書依然能夠將您對程式設計的深刻見解，帶到我的心中。**Ningjian Wang, Information System Engineer, The Vanguard Group**

《*Thinking in Java*》是本既優秀又具可讀性的書籍。我把它推薦給我所有的學生。**Dr. Paul Gorman, Department of Computer Science, University of Otago, Dunedin, New Zealand**

您打破「天下沒有白吃的午餐」這句不變的真理。而且不是那種施捨性質的午餐，是連美食家都覺得美味的午餐。**Jose Suriol, Scylax Corporation**

感謝有這個機會，看到這本書成爲一份傑作！在這個主題上，本書絕對是我所讀過的最佳書籍。**Jeff Lapchinsky, Programmer, Net Results Technologies**

您的這本書簡明扼要，易懂，而且讀起來心中充滿喜悅。**Keith Ritchie, Java Research & Development Team, KL Group Inc.**

的的確確是我所讀過最好的 Java 書籍！**Daniel Eng**

生平所見最好的 Java 書籍！**Rich Hoffarth, Senior Architect, West Group**

對於如此精采的一本好書，我應該向您道謝。遍覽各章內容，帶給我極大的樂趣。**Fred Trimble, Actium Corporation**

您肯定掌握了悠雅的藝術精髓，同時成功地讓我們對所有細節都心領神會。您也讓學習過程變得非常簡單，同時令人滿足。對於這麼一份無與倫比的入門書籍，我得向您致謝。**Rajesh Rau, Software Consultant**

《*Thinking in Java*》撼動了整個免費的世界！**Miko O'Sullivan, President, Idocs Inc.**

關於 *Thinking in C++*:

最好的書！
1995 年軟體開發雜誌（Software Development Magazine）Jolt Awared 得主。

本書成就非凡。您應該在架上也擺一本。其中的 iostreams 章節，是我所見表現最廣泛也最容易理解的。

Al Stevens
Contributing Editor, *Doctor Dobbs Journal*

Eckel 的這本書絕對是唯一如此清晰解說「如何重新思考物件導向程式發展」的一本書籍。也是透徹了解 C++ 的絕佳入門書。

Andrew Binstock
Editor, *Unix Review*

Bruce 不斷讓我對他的 C++ 洞察眼光感到驚奇。《*Thinking in C++*》則是他所有絕妙想法的整理沉澱。關於 C++ 的種種困擾，如果你需要清楚的解答，買下這本出眾的書就對了。

Gary Entsminger
Author, *The Tao of Objects*

《*Thinking in C++*》有耐心地、極具條理地探討了種種特性的使用時機與使用方式，包括：inline 函式、reference、運算子多載化、繼承、動態物件。也包括了許多進階主題，像是 template、exception、多重繼承的適當使用。整個心血結晶完全由 Eckel 對物件哲學與程式設計的獨到見解交織而成。是每個 C++ 開發者書架上必備的好書。如果您以 C++ 從事正式的開發工作，那麼《*Thinking in C++*》是您的必備書籍之一。

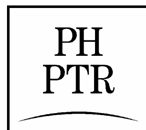
Richard Hale Shaw
Contributing Editor, PC Magazine

Thinking in Java

第二版

Bruce Eckel
President, MindView, Inc.

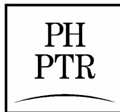
侯捷 / 王建興
合譯



Prentice Hall
Upper Saddle River, New Jersey 07458
www.phptr.com

```
Library of Congress Cataloging-in-Publication Data
Eckel, Bruce.
  Thinking in Java / Bruce Eckel.--2nd ed.
     p.   cm.
  ISBN 0-13-027363-5
  1. Java (Computer program language) I. Title.
 QA76.73.J38E25 2000
 005.13'3--dc21                        00-037522
                                           CIP
```

Editorial/Production Supervision: Nicholas Radhuber
Acquisitions Editor: Paul Petralia
Manufacturing Manager: Maura Goldstaub
Marketing Manager: Bryan Gambrel
Cover Design: Daniel Will-Harris
Interior Design: Daniel Will-Harris, www.will-harris.com



© 2000 by Bruce Eckel, President, MindView, Inc.

Published by Prentice Hall PTR
Prentice-Hall, Inc.
Upper Saddle River, NJ 07458

The information in this book is distributed on an "as is" basis, without warranty. While every precaution has been taken in the preparation of this book, neither the author nor the publisher shall have any liability to any person or entitle with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by instructions contained in this book or by the computer software or hardware products described herein.

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Prentice-Hall books are widely used by corporations and government agencies for training, marketing, and resale. The publisher offers discounts on this book when ordered in bulk quantities. For more information, contact the Corporate Sales Department at 800-382-3419, fax: 201-236-7141, email: *corpsales@prenhall.com* or write: Corporate Sales Department, Prentice Hall PTR, One Lake Street, Upper Saddle River, New Jersey 07458.

Java is a registered trademark of Sun Microsystems, Inc. Windows 95 and Windows NT are trademarks of Microsoft Corporation. All other product names and company names mentioned herein are the property of their respective owners.

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-027363-5

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada, Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Pearson Education Asia Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Bruce Eckel's

HANDS-ON JAVA™ SEMINAR

Learn the programming language of the World Wide Web

In this *step-by-step introduction* each carefully-chosen subject is covered in a lecture followed by hands-on programming exercises.

This course is for you if you can follow basic code examples written in C language syntax.

WWW.BRUCEECKEL.COM

Check www.BruceEckel.com
for in-depth details
and the date and location
of the next
Hands-On Java Seminar

- Based on this book
- Taught by Bruce Eckel
- Personal attention from Bruce Eckel and his seminar assistants
- Includes in-class programming exercises
- Intermediate/Advanced seminars also offered
- Hundreds have already enjoyed this seminar—

Bruce Eckel's Hands-On Java Seminar
Multimedia CD
It's like coming to the seminar!
Available at *www.BruceEckel.com*

- The *Hands-On Java Seminar* captured on a Multimedia CD!
- Overhead slides and synchronized audio voice narration for all the lectures. Just play it to see and hear the lectures!
- Created and narrated by Bruce Eckel.
- Based on the material in this book.
- Demo lecture available at *www.BruceEckel.com*

題獻

獻給那些正在努力創造
更偉大的下一代電腦語言的人們

綜覽（Overview）

譯序	xxix
Java 環境設定	xxxiii
序言（Preface）	1
簡介（Introduction）	9
01: 物件導論（Introduction to Objects）	29
02: 萬事萬物皆物件（Everything is an Object）	101
03: 控制程式流程（Controlling Program Flow）	133
04: 初始化與清理（Initialization & Cleanup）	191
05: 隱藏實作細目（Hiding the Implementation）	243
06: 重複運用 Classes（Reusing Classes）	271
07: 多型（Polymorphism）	311
08: 介面與內隱類別（Interfaces & Inner Classes）	349
09: 持有你的物件（Holding Your Objects）	407
10: 透過「異常」處理錯誤（Error Handling with Exceptions）	531
11: Java I/O 系統（The Java I/O System）	573
12: 執行期型別辨識（Run-time Type Identification）	659
13: 產生視窗和網頁小程式（Creating Windows & Applets）	689
14: 多緒（Multiple Threads）	825
15: 分佈式計算（Distributed Computing）	903
A: 物件的傳遞和回傳（Passing & Returning Objects）	1013
B: Java 原生介面（The Java Native Interface，JNI)	1065
C: Java 編程準則（Java Programming Guidelines）	1077
D: 資源（Resources）	1091
索引（Index ）	1099

目錄（What's Inside）

譯序	xxix
Java 環境設定	xxxiii
序言	**1**
第二版序	4
Java2	6
書附光碟（CD-ROM）	7
簡介	9
閱讀門檻	9
學習 Java	10
目標	11
線上說明文件	12
章節組織	13
習題	19
多媒體光碟	19
原始碼	20
撰碼標準	22
Java 版本	22
研討課程與顧問指導	23
關於錯誤	23
封面故事	24
致謝	25
Internet 上的貢獻者	28
1: 物件導論	**29**
抽象化的過程	30
每個物件都有介面	32
被隱藏的實作細節	35
重複運用實作碼	37
繼承：重複運用介面	38
「是一個」vs. 「像是一個」	42
隨多型而生的可互換物件	44
抽象基礎類別與介面	48
物件的形貌與壽命	49
群集器和迭代器 Collections and iterators	51
單根繼承體系 The singly rooted hierarchy	53
群集類別庫 Collection libraries 及其易用性支援	54
管家（housekeeping）面臨的兩難：誰該負責清理（clean-up）？	55
異常處理：面對錯誤的發生	57
多執行緒（Multithreading）	58
永續性（Persistence）	60
Java 與 Internet	60
Web 是什麼？	60
用戶端程式開發 Client-side programming	63
伺服端程式開發 Server-side programming	70
另一個截然不同的戰場：應用系統（applications）	71
分析與設計	71
階段 0：策劃	74
階段 1：建立什麼？	75
階段 2：如何建立？	79
階段 3：打造核心	83
階段 4：use cases 的迭代	84

階段 5：演化 85

取得成功 87

Extreme programming（極限編程） 88

測試優先 88

搭檔設計（Pair Programming） 90

Java 爲什麼成功 91

易於表達、易於理解的系統 91

透過程式庫發揮最大槓桿效應 92

錯誤處理 92

大型程式設計 92

過渡策略 93

實踐準則 93

管理上的障礙 95

Java vs. C++? 97

摘要 98

2: 萬事萬物皆物件 101

Reference 是操控物件之鑰 101

所有物件都必須由你建立 103

儲存在哪裡 103

特例：基本型別（primitive types） 105

Java 中的陣列 107

你再也不需要摧毀物件 107

生存空間（Scoping） 108

物件的生存空間 109

建立新的資料型別：class 110

資料成員（Fields）和函式（methods） 110

函式（methods）,引數（arguments）,回傳值（return values） 112

引數列（argument list） 114

打造一個 Java 程式 115

名稱的可視性 115

使用其他組件 116

關鍵字 **static** 117

初試啼聲 你的第一個 Java 程式 119

編譯與執行 121

註解及內嵌式文件 122

寓文件於註解 123

語法 124

內嵌的 HTML 125

@see: 參考其他的 classes 125

Class（類別）文件所用標籤 126

Variable（變數）文件所用標籤 127

Method（函式）文件所用標籤 127

文件製作實例 128

撰碼風格 129

摘要 130

練習 130

3: 控制程式流程 133

使用 Java 運算子 133

優先序（Precedence） 134

賦值、指派（Assignment） 134

數學運算子 Mathematical operators 137

遞增（increment）和遞減（decrement） 139

關係運算子（Relational operators） 141

邏輯運算子（Logical operators） 143

位元運算子（Bitwise operators） 146

位移運算子（Shift operators） 147

if-else 三元運算子 151

逗號運算子（comma operator） 152

應用於 **String** 身上的 operator + 153

使用運算子時的常犯錯誤 153

轉型運算子（Casting operators） 154
Java 沒有 "sizeof" 運算子 158
再探優先序（Precedence） 158
運算子綜合說明 159
流程控制 170
true 和 false 170
if-else 171
迭代（Iteration） 172
do-while 173
for 173
break 和 continue 175
switch 183
摘要 187
練習 188

4: 初始化與清理 191

以建構式（constructor）確保初始化的進行 191
函式多載化（method overloading） 194
區分多載化函式 196
搭配基本型別進行多載化 197
以回傳值作爲多載化的基準 202
Default 建構式 202
關鍵字 **this** 203
清理（Cleanup）：終結（finalization）與垃圾回收 207
finalize( ) 存在是爲了什麼？ 208
你必須執行清理（cleanup）動作 209
死亡條件（death condition） 214
垃圾回收器的運作方式 215
成員初始化（Member initialization） 219
指定初值 221
以建構式進行初始化動作 223
Array 的初始化 231
多維度 arrays 236
摘要 239
練習 240

5: 隱藏實作細目 243

package: 程式庫單元 244
獨一無二的 package 命名 247
自訂一個程式庫 251
利用 imports 來改變行爲 252
使用 package 的一些忠告 254
Java 存取權限飾詞（access specifiers） 255
"Friendly"（友善的） 255
public: 介面存取 256
private: 不要碰我 258
protected: 幾分友善 260
介面與實作（Interface and implementation） 261
Class 的存取權限 263
摘要 267
練習 268

6: 重複運用 Classes 271

複合（Composition）語法 271
繼承（Inheritance）語法 275
base class 的初始化 278
兼容複合及繼承 281
保證適當清理 283
名稱遮蔽（Naming Hiding） 286
複合與繼承之間的抉擇 288
protected（受保護的） 290
漸進式開發 291

向上轉型（Upcasting） 291
爲什麼需要向上轉型？ 293
關鍵字 **final** 294
Final data 294
Final methods 299
Final classes 301
最後的告誡 302
初始化以及 class 的載入 304
繼承與初始化 304
摘要 306
練習 307

7: 多型（Polymorphism） 311

再探向上轉型（Upcasting） 311
將物件的型別忘掉 313
竅門 315
Method-call（函式呼叫）繫結方式 315
產生正確的行爲 316
擴充性（Extensibility） 320
覆寫（overriding）vs. 重載（overloading） 324
Abstract classes（抽象類別）和 abstract methods（抽象函式） 325
建構式（Constructors）和 多型（polymorphism） 330
建構式叫用順序 330
繼承與 **finalize( )** 333
多型函式（polymorphic methods）在建構式中的行爲 337
將繼承（inheritance）運用於設計 339
純粹繼承（Pure inheritance） vs. 擴充（extension） 341
向下轉型（Downcasting）與執行期型別辨識（run-time type identification） 343
摘要 346
練習 346

8: 介面與內隱類別 Interfaces & Inner Classes 349

Interfaces（介面） 349
Java 的多重繼承 354
透過繼承來擴充 interface 358
產生常數群 359
將 interfaces 內的資料成員初始化 361
巢狀的（nesting）interfaces 362
Inner classes（內隱類別） 365
Inner classes（內隱類別）與 upcasting（向上轉型） 368
位於 methods 和 scopes 之內的 Inner classes（內隱類別） 370
匿名的內隱類別（Anonymous inner classes） 373
與外圍（outer）class 的連結關係 376
static inner classes（靜態內隱類別） 379
取用（referring）outer class 的物件 381
從多層巢狀 class 向外伸展觸角 383
繼承 inner classes 384
inner classes 可被覆寫嗎？ 385
Inner class 的識別符號（identifiers） 387
爲什麼需要 inner classes？ 388
Inner classes 和 control frameworks 394
摘要 402
練習 403

9: 持有你的物件 407

Arrays（陣列） 407
Arrays 是第一級物件 409

回傳一個 array 413

Arrays class 415

array 的充填（filling） 428

array 的複製（copying） 429

arrays 的比較 430

array 元素的比較 431

arrays 的排序（sorting） 435

在已排序的 array 中進行搜尋 437

array 總結 439

容器（container）簡介 439

容器的列印 441

容器的充填 442

容器的缺點：元素型別未定 450

有時候它總是可以運作 452

製作一個具有型別意識（type-conscious）的 **ArrayList** 454

迭代器（Iterators） 456

容器分類學（Container taxonomy） 460

Collection 的機能 463

List 的機能 467

根據 **LinkedList** 製作一個 stack 471

根據 **LinkedList** 製作一個 queue 472

Set 的機能 473

SortedSet 476

Map 的機能 476

SortedMap 482

Hashing 和 hash codes 482

覆寫 **hashCode( )** 492

持有 references 495

WeakHashMap 498

再論 Iterators（迭代器） 500

選擇一份適當的實作品 501

在各種 **List**s 之間抉擇 502

在各種 **Sets** 之間抉擇 506

在各種 **Maps** 之間抉擇 508

Lists 的排序和搜尋 511

公用函式（Utilities） 512

讓 **Collection** 或 **Map** 無法被更改 513

Collection 或 **Map** 的同步控制 514

未獲支援的操作 516

Java 1.0/1.1 的容器 519

Vector 和 Enumeration 519

Hashtable 521

Stack 521

BitSet 522

摘要 524

練習 525

10: 透過異常（Exceptions）處理錯誤 531

基本異常（Basic exceptions） 532

異常引數（Exception arguments） 533

異常的捕捉 534

try block 535

異常處理常式（Exception handlers） 535

撰寫你自己的異常類別 537

異常規格（exception specification） 542

捕捉所有異常 543

重擲（Rethrowing）異常 545

Java 標準異常 549

RuntimeException 的特殊情況 550

以 finally 進行清理（cleanup） 552

爲什麼需要 **finally**？ 554

缺憾：異常遺失 557

異常的侷限 558

建構式（Constructors） 562

異常的比對（matching） 566

異常的使用原則 568

摘要 568

練習 569

11: Java I/O 系統 573

File class 574

目錄列示器（a directory lister） 574

目錄的檢查和建立 578

輸入和輸出 581

InputStream 的類型 581

OutputStream 的類型 583

附加屬性（attributes）和有用介面 585

透過 **FilterInputStream** 自 **InputStream** 讀取資料 586

透過 **FilterOutputStream** 將資料寫入 **OutputStream** 587

Readers 和 **Writers** 589

資料的來源（sources）和去處（sinks） 590

改變 stream 的行爲 591

未曾有任何變化的 classes 592

RandomAccessFile 593

I/O streams 的典型運用 594

Input streams 597

Output streams 599

這是個臭蟲嗎？ 601

管線化的（piped）streams 602

標準 I/O 602

讀取標準輸入 603

將 **System.out** 轉換爲 **PrintWriter** 604

標準 I/O 重導向（Redirecting） 604

壓縮 606

運用 GZIP 進行單純壓縮 607

運用 Zip 儲存多份檔案資料 608

Java ARchives (JARs) 611

物件次第讀寫（Object serialization） 613

找出 class 618

控制次第讀寫（serialization） 619

使用物件永續機制（persistence） 630

被「語彙單元化」（Tokenizing）的輸入動作 639

StreamTokenizer 639

StringTokenizer 642

檢驗大小寫 645

摘要 655

練習 656

12: 執行期型別辨識 Run-time Type Identification 659

爲什麼需要 RTTI 659

Class 物件 662

轉型之前先檢查 665

RTTI 語法 674

Reflection: 執行期類別資訊 677

實作一個函式提取器（extractor） 679

摘要 685

練習 686

13: 製作視窗和網頁小程式（Applets） 689

基本的 applet 692

Applet 的束縛 692

Applet 的優點 693

應用程式框架（Application frameworks） 694
在 Web 瀏覽器上執行 applets 695
運用 Appletviewer 698
測試 applets 698
從命令列執行 applets 700
一個顯示框架 702
運用 Windows Explorer 705
製作一個按鈕（button） 706
捕捉一個事件（event） 707
文字區（Text areas） 711
控制版面佈局 712
BorderLayout 713
FlowLayout 714
GridLayout 715
GridBagLayout 716
絕對定位 716
BoxLayout 717
最好的方法是什麼？ 721
Swing 的事件模型（event model） 722
事件（event）及監聽器（listener）的種類 723
追蹤多種事件 730
Swing 組件一覽 734
按鈕（Buttons） 734
圖示（Icons） 738
工具提示（Tool tips） 740
文字欄（Text fields） 740
框線（Borders） 743
JScrollPanes 744
迷你文字編輯器（mini-editor） 747
方鈕，核取方塊（Check boxes） 748
圓鈕（Radio buttons） 750
複合方塊，下拉式清單 (Combo boxes, drop-down lists) 751
清單方塊（List boxes） 753
頁籤式嵌版（Tabbed panes） 755
訊息方塊（Message boxes） 756
功能表（Menus） 759
冒起式功能表（Pop-up menus） 766
繪圖（Drawing） 768
對話方塊（Dialog Boxes） 771
檔案對話方塊（File dialogs） 776
Swing 組件上的 HTML 779
滾軸（Sliders）和進度指示器（progress bars） 780
樹狀組件（Trees） 781
表格（Tables） 784
選擇外觀風格（Look & Feel） 787
剪貼簿（clipboard） 790
將 applet 封裝於 JAR 檔 793
編程技術 794
動態繫結事件（Binding events dynamically） 794
將 business logic 和 UI logic 隔離 796
標準型式 799
視覺化程式設計與 Beans 800
什麼是 Bean？ 801
運用 **Introspector** 提取出 **BeanInfo** 804
一個更為複雜精巧的 Bean 811
Bean 的包裝（Packaging a Bean） 816
Bean 所支援的更複雜功能 818
其他 819
摘要 819
練習 820
14: 多緒（Multiple Threads） 825
反應靈敏的 UI 826
繼承自 **Thread** 828

運用執行緒打造出反應靈敏的 UI 831
結合執行緒和程式主類別 834
產生多個執行緒 836
Daemon 執行緒 840
共享有限資源 842
不當的資源存取 842
Java 如何共享資源 848
再探 JavaBeans 854
停滯（阻塞, Blocking） 859
轉爲停滯狀態（Becoming blocked） 860
死結（Deadlock） 872
優先權（Priorities） 877
優先權的讀取和設定 878
執行緒群組（Thread groups） 882
再探 **Runnable** 891
過多的執行緒 894
摘要 899
練習 901
15: 分佈式計算（Distributed Computing） 903
網絡編程 904
機器的識別 905
Sockets 909
服務多個用戶 917
資料元（Datagrams） 923
在 applet 中使用 URL 923
更多的網絡相關資訊 926
JDBC, Java 資料庫連結機制 927
讓本節實例正常運作 931
一個 GUI 版的查詢程式 935
爲什麼 JDBC API 看起來如此複雜 938
一個更複雜的範例 939
Servlets 948
Servlet 基本教練 949
Servlets 和多緒 954
以 servlets 處理 sessions 955
執行 servlet 範例程式 960
Java Server Pages（JSP） 960
隱式物件（Implicit objects） 962
JSP 指令（directives） 963
JSP 腳本描述成份 964
取出欄位（fields）和數值（values） 966
JSP 的頁面屬性（page attributes）和有效範圍（scope） 968
處理 JSP 中的 sessions 969
產生並修改 cookies 971
JSP 摘要 972
RMI (Remote Method Invocation) 遠端函式調用 973
遠端介面（Remote interfaces） 973
實作出遠端介面 974
產生 stubs 和 skeletons 978
使用遠端物件（remote object） 979
CORBA 980
CORBA 的基本原理 981
一個實例 983
Java Applets 和 CORBA 989
CORBA vs. RMI 989
Enterprise（企業級）JavaBeans 990
JavaBeans vs. EJBs 991
EJB 規格 992
EJB 組件 993
EJB 組件成份 994

EJB 的各項操作 995
EJBs 的類型 996
分發（Developing）EJB 997
EJB 摘要 1003
Jini: 分佈式服務 1003
Jini 的來龍去脈 1003
什麼是 Jini? 1004
Jini 如何運作 1005
Discovery 動作 1006
Join 動作 1006
Lookup 動作 1007
介面和實作的分離 1008
將分佈式系統抽象化 1009
摘要 1010
練習 1010

A: 物件的傳遞和回傳 1013

References 的傳遞 1014
別名（Aliasing） 1014
製作一個區域性副本 1017
Pass by value（傳值） 1018
物件的克隆（Cloning） 1018
賦予 class 克隆能力 1020
成功的克隆 1022
Object.clone( ) 的效應 1025
克隆一個複合物件（composed object） 1027
對 **ArrayList** 進行深層拷貝 1030
透過 serialization 進行深層拷貝 1032
將克隆能力加到繼承體系的更下層 1034
爲什麼要有這種奇怪的設計？ 1035
克隆能力（cloneability）的控制 1036
copy 建構式 1042
唯讀類別（Read-only classes） 1047
撰寫一個唯讀類別 1049
恆常性（immutability）的缺點 1050
恆常不變的 **Strings** 1052
String 和 **StringBuffer** 1056
Strings 是特殊的東西 1060
摘要 1060
練習 1062

B: Java 原生介面 Java Native Interface (JNI) 1065

原生函式的調用 1066
表頭檔產生器：javah 1067
名稱重整（Name mangling）與函式標記（function signatures） 1068
實作出你自己的 DLL 1068
取用 JNI 函式: 透過 **JNIEnv** 引數 1069
存取 Java Strings 1071
傳遞和運用 Java 物件 1071
JNI 和 Java 異常 1074
JNI 和多緒 1075
使用既有的程式碼 1075
補充資訊 1076

C: Java 編程準則 1077
設計 1077
實作 1084
D: 資源 1091
軟體 1091
書籍 1091
分析& 設計 1093
Python 1095
我的著作 1096
索引 1099

譯序 (1)

侯捷

我完成 942+ 頁的《深入淺出 MFC》2/e（著作）和 1237+ 頁的《*C++ Primer 3e* 中文版》（譯作）之後，曾經罹患一種厚書恐懼症。畢竟大部頭書籍的寫作和翻譯都異常艱巨，包括材料之取捨、用語風格及術語之協調、長時間的工作過程和期待…，對於作者和譯者的組織力、創作力、毅力都是一種嚴苛的考驗。

但《*Thinking in Java*》2e 畢竟不同一般。這本書將在 Java programming 基礎教育上起重要作用。我於是懷著戒慎恐懼（但也開心）的心情再度接下這份重任。建興和我，以幾近一年的時間，完成了這本 1127+ 頁大部頭書籍的翻譯和版面製作。

Java 的面向太廣太廣，不可能有一本涵蓋全貌的書籍（我們也不應該有「一次買足」的心理）。就相對全貌而言，《*Thinking in Java*》是一本取材廣泛而表現優異的作品。其最大特色就是：(1) 內容涵蓋面大，(2) 許多主題（特別是物件導向編程技術）極爲深入，廣泛而又深刻的論述形成了 1127+ 頁的份量，(3) 英文版由作者 Bruce Eckel 免費開放，造福很多人，也因而影響了很多人（因此你很容易找到一個可以共同討論書中觀點的朋友）。

本書的優異表現，從讀者的熱情回應（摘列於書前）可見一斑。乍看之下這雖然像是一本初學教本，而它也的確勝任此一角色，但它對某些主題（例如 Polymorphism, Object Serialization, Reflection, RTTI, I/O, Collections）的深入討論，肯定也能帶給許多 Java 老手新的刺激。

本書（繁體中文版）延用我個人喜愛的「頁頁對譯」方式，用以和英文版頁次達成一種直覺對應，並輕鬆保留書內所有交叉參考（cross reference）和書後完整索引。索引詞條皆不譯，我個人認為值得保留的英文術語亦不譯（第一次出現或某個頻率下我會讓它英中並陳）。之所以保留英文術語，我已多次為文闡述個人想法，文章都公佈在侯捷網站上（http://www.jjhou.com）。

做為本書英文版的一個認真讀者，我要說，我確實從這本書學習了許多深刻的見識。希望透過我和建興的努力，這本中文版能夠協助更多人高效、精準、深刻地認識並學習 Java。

為了仿效並致敬 Bruce Eckel 開放《*Thinking in Java*》2e 英文版的精神，侯捷網站（http://www.jjhou.com）將開放本中文版前 9 章及 4 個附錄，佔全書篇幅幾近 1/2。我要特別感謝碁峰出版公司對此一決定的鼎力支持。

今年我同時準備了初階、中階、高階共四本 Java 書籍中譯本以饗讀者，它們是：

1. *Thinking in Java*, 2e, by Bruce Eckel, Prentice Hall, 2000
2. *Practical Java,* by Peter Haggar, Addison Wesley 2000
3. *Effective Java,* by Joshua Bloch, Addison Wesley 2001
4. *Refactoring - Improving the Design of Existing Code*, by Martin Fowler, Addison Wesley 2000（此書不限 Java，只是以 Java 為表述工具）

侯捷 2002.07.01 臺灣.新竹

`http://www.jjhou.com`（中文繁體）
`http://jjhou.csdn.net`（中文簡體）
`jjhou@jjhou.com`（電子郵箱）

- 中文版勘誤維護於侯捷網站 http://www.jjhou.com。
- 本書英文版前後數頁關於作者 Bruce Eckel 個人網站、研討會課程、CD 產品廣告（圖片和文字）皆不譯，但保留，以示對作者之尊重。

譯序(2)

王建興

呼，好厚的一本書，可不是嗎？

能夠參予這本書籍的翻譯是我的榮幸。以 Java 入門書而言，《*Thinking in Java*》一直都是我心目中的第一首選。Java 的世界看似複雜其實單純。炫人耳目的反倒是核心之外的各種 library。對於一本入門書來說，首要之務無非是讓讀者了解語言核心，而《*Thinking in Java*》成功地做到了這一點。它在內容篇幅的拿捏上恰到好處，讓讀者深入了解語言核心的同時，又能夠對 Java 所涵蓋極爲廣泛的各套 API，都能夠有所認識。這是它之所以能夠成爲如此成功的一本書的原因。

這樣的好書，當然值得花這麼多的時間來和它相處。

謝謝侯大哥給予機會參與這本譯作的產生，以及長久以來的指導。也謝謝 CSZone 上的 sofar 以及我的妹妹嘉凌，與他們的討論解決了許多我對本書內容的疑難。當然要謝謝我的媽媽，不時提醒我要加緊腳步。最後要謝謝淑卿，允許我撥出原本應當屬於她的時間來進行本書的翻譯工作。謝謝大家。

王建興@新竹

關於本書術語，請注意：

- 侯捷網站提供一份計算機術語英中繁簡對照（持續完善中），請參考。
- Java 所謂之 method（方法），即 C/C++ 之 member function（成員函式）。將 method 譯爲「方法」，容易與前後文混雜，不易突顯術語之獨特性，因此本書一律將 method 譯爲「函式」或「成員函式」。
- Java 所謂之 field（資料欄），即 C/C++ 之 data members（資料成員）。由於 field 一詞在其他諸多地方（例如資料庫）也被採用，爲避免混淆並求一致性，本書一律將 class field 譯爲「資料成員」。
- copy 和 clone 在臺灣皆謂之「複製」。然而其中有些不同。clone, cloneable, cloneability 且爲一種 Java 技術概念。因此，爲求區別，本書將 copy 譯爲「複製」或「拷貝」，將 clone 譯爲「克隆」（音譯，取乎「拷貝」之譯法。見 p1018）。「克隆」在中國大陸是一個被普遍使用的詞。

Java 環境設定

by 侯捷

本附錄為譯者所加，說明 Java 開發工具（JDK）之下載及環境設定。

任何人可於 Java 官方網站（http://java.sun.com）自由下載 JDK，如圖 1。

圖 1. Java 官方網站提供 JDK 最新版本

下載所得是個可執行檔，一旦被執行起來，便自動解壓縮並進行安裝。安裝程序很簡單，只要遵循畫面指示進行即可。預設安裝路徑是 c:\jdk1.3 或 c:\jdk1.4.0（或類似路徑）。

安裝 JDK 1.3 後，C:\JDK1.3\BIN\ 內將放置各種開發工具，部分如下：

jar.exe	壓縮工具。請參考 p611。
java.exe	執行工具，可接受.class 檔（byte code）並啓動 Java 虛擬機器執行之。請參考 p122。
javac.exe	編譯器（其實只是個外包器，wrapper）。可接受.java 檔（程式原始碼）並產出.class 檔（byte code）。請參考 p122。
javadoc.exe	文件製作工具。請參考 p123。
javah.exe	表頭檔（header）產生器。請參考 p1067。
appletviewer.exe	網頁小程式（applet）執行器。請參考 p698。
...	

我個人習慣爲 JDK 撰寫一個環境設定批次檔，如以下之 jdk13.bat：

```
@echo off
rem JDK1.3
set PATH=C:\jdk1.3\bin;C:\WINDOWS;C:\WINDOWS\COMMAND
set classpath=.;c:\jdk1.3\lib\tools.jar
```

將這份環境設定檔設爲某個 DOS 視窗的「內容」表單下的「程式」附頁中的批次檔，如圖 2。於是，一旦開啓（雙擊）該 DOS 視窗，就會自動設定好上述的 JDK1.3 開發環境和執行環境，如圖 3。

一個 Java 程式可由多個 .java 檔（源碼）構成，每個 .java 檔內可容納多個 classes。編譯器針對每個 class 產生對應而獨立的 .class 檔(內置 byte code)。凡 public class 並含有 `main()` 函式者，該函式便可被當做程式進入點，而該 class 名稱也就是你應該在 **java.exe** 命令列中指定的名稱（大小寫必須完全一致）。請見 p121「編譯與執行」。

圖 2. 設定 DOS 視窗，使之執行前述的 jdk13.bat，將 JDK 環境設定妥當。

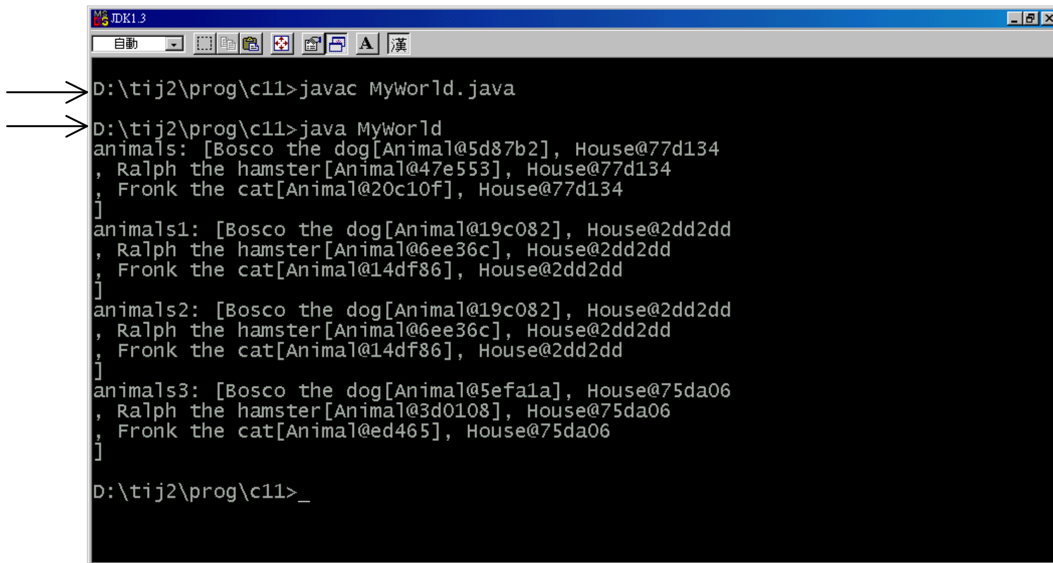


圖 3. 開啓(雙擊)圖 2 設定之 DOS 視窗，便可在其中編譯 Java 程式。本畫面顯示以 **javac.exe** 編譯本書第 11 章程式 MyWorld.java（將獲得可執行檔 MyWorld.class）。再以工具 **java.exe** 執行 MyWorld.class（命令列中不可指定副檔名 ".class"）。

引言

我想起我的兄弟 Todd，他正從硬體領域大躍進到程式設計領域。基因工程將是下一個大革命的戰場。

我們將擁有許多被設計用來製造食物、燃料、塑膠的各類微生物；這些微生物的出現，可以免去污染，讓我們僅付出遠少於現今所付的代價，便足以主宰整個物質世界。我原以爲，這場革命規模之大，將使電腦革命相形見絀。

然而，後來，我發覺我犯了科幻作家所犯的同樣錯誤：對科技的迷失（當然，這在科幻小說中屢見不鮮）。有經驗的作家都知道，故事的重點不在技術，而在於人。基因工程無疑會對我們的生活造成極大衝擊，但是我不確定它是否會阻礙電腦革命（電腦革命也帶動了基因工程的到來）或至少說是資訊革命。所謂資訊，關注的是人際之間的溝通。是的，車子、鞋子、尤其是基因治療，都很重要，但這些東西最終都和陷阱沒什麼兩樣。人類與世界共處的方式，才是真正關鍵所在。而這其中，溝通居重要角色。

本書是一個例子。大多數人認爲我很大膽、或者說有點瘋狂，因爲我把所有材料都放上了 Web。「那麼，還有什麼購買理由呢？」他們這樣問我。如果我的性格保守謹慎一些，我就不會這麼做了。但是我真的不想再用同樣老舊的方式來撰寫一本新的電腦書籍。我不知道會發生什麼事，但事實證明，這是我對書籍所做過的最棒的一件事。

首先，人們開始把校正結果送回。這是個讓人驚嘆的過程，因爲人們仔細端詳每個角落、每個縫隙，揪出技術上和文法上的種種錯誤，讓我得以更進一步消除所有毛病，而這些毛病原本是被我遺漏未察的。人們往往對這種作法表示驚恐，他們常常說「唔，我不確定我這麼做是否過於吹毛求疵…」，然後扔給我一大堆錯誤。我發誓，我自己從未曾察覺這些錯誤。我很喜歡這種群體參與的過程，這個過程也使這本書更加獨特。

但是，接著我開始聽到「嗯，很好。整個電子版都能夠放到網絡上實在不錯，可是我希望擁有出版商印出來的、裝訂成冊的紙本。」我曾經努力讓每個人能夠更輕鬆地以美觀的方式把它印出來，但這麼做似乎還是無法滿足要求此書付梓的強大呼聲。大部份人都不希望在電腦螢幕上讀完一整本書，也不希望總是拖著一捆捆的紙。即便這些紙張印刷再美觀，對他們來說也不具半點吸引力（而且我想雷射印表機的碳粉也不便宜）。即使是電腦革命，似乎也難以削減出版商的生意。不過，某個學生提出了一種看法，他認爲這也許會成爲未來的一種出版模型：先在 Web 上公開書籍內容，只有在吸引了足夠關注時，才考慮製作紙本形式。目前，絕大多數書籍都不賺錢，這種作法或許可以讓整個出版產業更有利潤一些。

這本書以另一種形式對我產生了啓迪。一開始我把 Java 定位在「只不過是另一個程式語言」。從許多角度看的確如此。但是隨著時間流逝，對 Java 的學習日深，我開始明白這個程式語言的最根本目的，和其他我所見過的程式語言有著極大的不同。

程式設計，就是對複雜度的控管。複雜度包括：待解問題的複雜度和底層機器的複雜度。因爲有著這樣子的複雜度，所以大多數程式開發專案都失敗了。到目前爲止，所有我所知道的程式語言，沒有一個竭盡所能地將主要設計目標鎖定在「克服程式發展與維護過程中的種種複雜度」上[1]。當然，許多程式語言當初設計時也曾將複雜度考慮進去，但是總有被視爲更本質的問題混雜進來。毫無疑問，那些問題也都是會讓語言使用者一籌莫展的問題。舉例來說，C++回溯相容於 C（俾使熟悉 C 的程式員得以比較

[1] 在此第二版中，我要收回這句話。我相信，Python 語言極爲逼近這個目標。請參考 *www.Python.org*。

輕鬆地跨越門檻），並具備高執行效率的優點。這兩個特性都是大有幫助的目標，並擔負起 C++成敗的重責大任。不過，兩者也帶來了額外的複雜度，使得某些專案無法完成。（通常你可以將此點歸罪於程式開發人員或管理人員，不過如果某個語言可以協助我們捕捉錯誤，何樂不為？）Visual Basic（VB）是另一個例子，它被 BASIC 這個「其實不以擴充性為設計目標」的語言侷限住，使得所有強硬堆累於 VB 之上的擴充功能，都造成了可怕至極且難以維護的語法。Perl 也回溯相容於 Awk、Sed、Grep、以及其他諸般它想取代的 Unix 工具，於是衍生出諸如「能寫卻不能讀」的程式碼（write-only code）這種指責（意思是，寫完之後的數個月內，你都還是無法閱讀它）。另一方面，C++、VB、Perl、Smalltalk 之類的程式語言，都在複雜度議題上有著相當程度的著墨，十分成功地解決了某些類型的問題。

當我了解 Java 之後，最叫我感動的，莫過於 Java 似乎把「為程式員降低複雜度」做為一個堅定的目標。就好像是說：『我們的唯一考量，就是要降低穩固程式碼的產生時間和困難度』。早期，這個目標只開花結果於程式碼的撰寫上，但這些程式碼卻無法執行得很快（雖然目前有許多保證，承諾 Java 總有一天能夠執行得多快多快）。但是 Java 的確出人意表地大幅縮短了發展時間；比起發展等價的 C++ 程式來說，大概只需一半或甚至更少時間。光是這個結果，就足以省下驚人的時間與金錢。不過，Java 並未停止腳步。Java 繼續將多執行緒、網絡程式設計等等複雜而益形重要的工作包裝起來。透過語言本身的性質以及程式庫（libraries），有時能夠使這些工作變得輕而易舉。最後一點，Java 著眼於某些有著極高複雜度的問題：跨平台程式、動態程式碼改變、安全議題，其中每一個問題都能夠套用在整個複雜度頻譜的任何一點上。所以儘管有著眾所周知的效率問題，Java 帶來的保證卻是很棒的：可以讓我們成為具有高生產力的程式開發者。

我所見到的種種巨幅改變，有一些是發生在 Web 身上。網絡程式設計一直都不是件簡單的事，但是 Java 把它變簡單了（而 Java 語言設計者仍舊繼續努力使它變得更簡單）。網絡程式設計所討論的，便是讓我們以更有效率、成本更低的方式，和其他人通訊，超越舊式電話媒介（單是電子郵件

便已經革命性地改變了許多事情）。當我們能夠在與他人溝通這件事情上著力更多，神奇的事情便會開始發生，或許甚至比基因工程所許諾的遠景，更讓人感到神奇。

透過所有努力 – 程式開發、團隊開發、使用者介面的打造（讓程式可以和使用者互動）、跨平台的執行、跨 Internet（網際網、互聯網）通訊程式開發的大量簡化 – Java 擴展了「人際之間」的通訊頻寬。我認為，通訊革命的成果或許不應只是圍繞在傳輸頻寬提高後所產生的效率上打轉；我們將會看到貨真價實的革命，因為我們能夠更輕鬆地和其他人溝通：可以是一對一的形式、可以是群體通訊的形式、也可以是和全地球人通訊的形式。我曾經聽人主張，接下來的革命會是一種全球意志的形成，來自於足夠數量的人們和足夠數量的相互聯繫。Java 可能是、也可能不是點起這把撩原之火的星焰，但至少存在著可能。這使我覺得，這個語言的教學是一件有意義的事。

第二版序

關於本書的第一版，人們給了我許多許多精彩的意見。我當然對此感到非常愉快。不過有時候讀者也有抱怨。一個常常被提起的抱怨便是：這本書太大了。對我來說，如果「頁數過多」的確是你唯一的怨言，那真是一個讓人無言以對的責難。（這讓人想起奧地利皇帝對莫札特作品的非難：『音符太多了！』當然我並不是將自己拿來和莫札特相提並論）我只能夠假設，這樣的怨言出自於一個「被塞進太多 Java 語言的廣闊內容，而還未看過其他相同主題的書籍」的人。就拿我最喜歡的一本參考書來說好了，它是 Cay Horstmann 和 Gary Cornell 合著的《*Core Java*》（Prentice-Hall 出版），其規模甚至大到必須拆成兩冊。儘管如此，在這一（第二）版中，我努力嘗試的事情之一，便是修剪過時的內容，或至少不是那麼必要的內容。這麼做我覺得很自在，因為原先內容仍然擺在網站 *www.BruceEckel.com* 上，而且本書所附光碟中也有第一版的內容。

如果你想取得原本內容，沒有任何問題，這對作者而言也是一種欣慰。舉個例子，您可能會發現第一版的最後一章「專案（Projects）」已不復存在；有兩個專案被整合到其他章節裡頭，因此剩餘部份也就變得不再適合存在。同樣的，「設計樣式（Design Patters）」這一章的內容變得更豐富了，因此也獨立成冊了（可自我的網站下載）。基於這些原因，本書理應變得苗條一些。

事實卻非如此。

最大的問題是，Java 語言本身仍在持續發展當中，企圖爲所有可能用到的東西提供標準化介面，因此 API 的數量不斷擴增。所以就算看到 JToaster 這樣的 API 出現，我也不會吃驚）（譯註：toaster 是烤麵包機，JToaster 意指標準化的烤麵包機類別。連烤麵包機都有標準類別，藉此誇飾 Java 2 所涵蓋的類別的廣泛程度）。「涵蓋所有 API」似乎逾越本書範圍，應該交給其他作者完成，但儘管如此，仍然有某些議題難以略去。伺服端 Java（主要是 Servlets 和 Java Server pages，*JSPs*）便是這些議題中涵蓋最廣的一個。伺服端 Java 無疑爲 WWW 的諸多問題提供了出色的解決方案。尤其當我們發現「現存許多不同的 Web 瀏覽器平台，無法提供一致的用戶端程式開發」時，更顯其重要性。此外 Enterprise Java Beans（EJBs）的問世，也讓人們希望更輕易地發展資料庫連結、交易處理、安全考量。這些問題在本書第一版本中通通被放進「網絡程式設計」一章。本版則改用了一個對每個人來說都愈來愈不可或缺的名稱：「分佈式計算」。你還會發現，本書同時也將觸角伸至 Jini（讀作 genie，這真的只是個名字，不是什麼字首縮寫）的概要性說明。此一尖端技術讓我們重新思考應用程式之間的接駁形式。當然，本書所有內容都已改用 Swing 這個 GUI 程式庫。再次提醒你，如果你想要取得原有的 Java 1.0/1.1 書籍內容，可自 *www.BruceEckel.com* 免費下載（網站上同時也包含第二版書附光碟的內容；新的材料還會陸續增加）。

本書不但新增的少數 Java 2 語言特性，也徹頭徹尾地翻修了一遍。最主要的改變是第九章的群集（collection），我將重點改放在 Java 2 的群集，並修繕本章內容，更深入鑽進某些更重要的群集議題之中，尤其是對 hash function（雜湊函式）運作方式的討論。此外還有其他一些變動，包括重寫

第一章，抽掉部份附錄以及我認爲新版不再需要的內容。整體而言，我重新審閱所有內容，移去第二版中不再需要的部份（但仍保留它們的電子版於我的網站上），並將新的更動加入，然後儘可能改進我能夠改進的所有地方。這種大變動是由於語言本身持續有所變化 － 即使不再像以前那麼大刀闊斧。以此觀之，毫無疑問，本書還會有新版面世。

對那些吃不消本書厚度的讀者們，我向你們致歉。不管你們是否相信，我真的傾盡全力地想辦法讓這本書更輕薄。儘管體積依舊龐大，我認爲有其他令你滿足的方式。本書也有電子版形式（不論是從網站取得，或是從本書所附光碟取得），你可以帶著你的筆記電腦，把這本電子書放進去，不增加絲毫重量。如果你希望更輕便地閱讀，也可以找到四處流傳的 Palm Pilot 版（有人跟我說，他總是躺在床上以 Palm 閱讀本書內容，並打開背光裝置（譯註：Palm PDA 上一種用於夜間燈光不足時的顯示模示）以免打擾太太。我想這應該有助於讓他早點進入夢鄉）。如果你一定得在紙上閱讀，我知道有些人一次印出一章，然後帶在公事包裡頭，在電車上閱讀。

Java 2

本書撰寫之際，Sun 即將發行 *Java Development Kit*（JDK）1.3，並預告 JDK 1.4 將有的更動。雖然這些版本的主號碼還停留在 1，但是 "Java2"卻是 JDK 1.2 及其後續版本的標準名稱。這突顯了介於「老式 Java」（其中有許多本書第一版中抱怨過的缺點）與新版本之間的重大差異。在這些更新更精進的新版本中，缺點變少了，加入了許多新特性和美妙的設計。

本書是爲 Java2 而寫。能夠擺脫過去的沉舊內容，重新以更新的、更精進的語言來寫作，這個事實給我憑添許多樂趣。舊有的資訊依舊存於網站上的第一版電子書及光碟中，如果你使用 Java 2 之前的版本，便可拿它們來參考。由於每個人都可以自 java.sun.com 免費下載 JDK，所以就算我

以 Java 2 撰寫本書，也不會讓讀者爲了升級而付出金錢。

不過，還是有點小麻煩，這是因爲 JDK 1.3 提供了某些我想用到的改進功能，而目前 Linux 上正式發行的版本只到 JDK1.2.2。Linux（見 *www.Linux.org*）是個和 Java 關聯極深的重要開發環境。Linux 正以極高的速度成爲最重要的伺服器平台，快速、穩定、強固、安全、容易維護，而且免費，的確是電腦史上的革命（我不認爲可以在過去任何工具上同時看到這些特色）。Java 則在伺服端找到了極爲關鍵的利基所在，也就是 Servlets 這個爲傳統 CGI 程式設計帶來巨大改進效應的新技術（本主題含括於「分佈式計算」一章）。

所以，雖然我想完全使用最新功能，但是讓每一個程式都在 Linux 上順利編譯，對我而言至關重要。當你解開本書原始碼檔案，並在 Linux 上以最新的 JDK 進行編譯，你會發現所有程式應該都能夠順利完成。然而你會看到我四處放置的有關 JDK 1.3 的註記。

書附光碟

這個（第二）版本提供了一份紅利：附於書後的光碟。過去我不想將光碟放在我所撰寫的書籍後面，原因是我覺得，爲了大小僅數百 K bytes 的原始碼，用了那麼大一片光碟，實在說不過去。我傾向於讓讀者從我的網站下載這些東西。但是現在，你看到了，本書所附的光碟大有不同。

這片光碟當然包含了本書所有原始碼，此外還包含本書完整內容的多種電子形式。我最喜歡 HTML 格式，不但速度快，索引也十分完備 － 只要在索引或目錄上按下某個項目，馬上就可以跳到你想閱讀的地點。

整張光碟有著超過 300 Mega 的滿滿內容，是一個名爲《*Thinking in C: Foundations for C++ & Java*》的全多媒體課程。我原本委託 Chuck Allison 製作這張光碟，是希望做成一個獨立產品。但後來決定將它放在

《*Thinking in C++*》和《*Thinking in Java*》兩本書的第二版中，因爲持續有一些尚未具足 C 語言背景的人來上我的研討課程，他們認爲：『我是個聰明的傢伙，我不想學 C，只想學 C++或 Java，所以我直接跳過 C，進到 C++/Java 裡頭。』加入研討班後，這些人才逐漸了解，認識基本的 C 語法，是多麼重要。將光碟片納入本書，我便可以確信，每個人都能夠在準備充裕的情形下參加我的研討課程。

這份光碟同時也讓本書得以迎合更多讀者的喜好。即使本書第三章「程式流程的控制」已經涵蓋了取自 C 核心部份的 Java 相關語法，這張光碟仍然是一份很好的導引。它所假設的學員基礎也比本書寬鬆許多。希望這張光碟的附上，能夠使愈來愈多的人被帶領進入 Java 程式設計的大家庭。

簡介

Introduction

Java，一如人類所使用的任何一種自然語言，提供的是意念表達的機制。如果使用恰到好處，那麼作爲一種表達媒介，當你打算解決的問題益形龐大複雜，解法益形簡易而富彈性。

你不能僅僅將 Java 看成某些特性的集合 — 這些特性之中某些被支解而被獨立對待時，將不具絲毫意義。只要你心中存有「設計」念頭，而非單單只是撰碼，那麼便可以整體性地運用 Java 的所有組成。如果想以這種方式來了解 Java，你還必須了解其衍生問題，以及在一般情況下其程式設計過程所衍生的問題。本書所討論的是程式設計的問題、它們之所以成爲問題、以及 Java 採取的解決方案。因此，我在各章之中所描述的各種特性，其建構基礎皆是「我所看到的此一程式語言，在解決特定問題時的方式」。透過這種方式，希望能夠對你潛移默化，漸漸地讓 Java 式思考模式成爲對你而言再自然不過的一件事。

我的態度始終一致：你得在腦中建立模型，藉以發展出對此程式語言的深層體會和洞悉；如果你在某個問題上陷入思想泥沼，可將它饋入腦內模型，並推導答案。

閱讀門檻

本書假設你對編程（programming）一事有著某種程度的熟悉：你已經了解程式是許多敘述句的集合，你已經明白副程式/函式/巨集的觀念，懂得諸如 "if" 之類的流程控制敘述，以及諸如 "while" 之類的迴圈概念…等等。你可以從許多地方學習到這些東西，例如用巨集語言來設計程式，或使用諸如 Perl 之類的工具來工作。只要你在編程上的境界能夠「自在地以編程基本概念來編寫程式」，那麼你便可以閱讀本書。當然，本書對於 C 程式

員而言，相對容易些，對 C++程式員而言更是如此。但如果你對這兩種語言毫無經驗，也不必妄自菲薄（但請懷著高度的學習熱忱。本書所附的多媒體光碟，能夠帶領你快速學習對 Java 而言必備的基本 C 語法）。我同時會引入物件導向程式設計（Object-Oriented Programming，OOP）的觀念，以及 Java 的基本控制機制。你會接觸到這一切，並在第一個練習題中練習基本的流程控制敘述。

雖然書中有許多參考資料，介紹的是 C/C++語言的特性，但那並不是爲了進行更深層的詮釋，而只是希望幫助所有程式員正確看待 Java，畢竟 Java 是 C/C++的後裔。我會儘量簡化這些參考資料，並解釋所有我認爲 non-C/C++ 程式員可能不怎麼熟悉的部份

學習 Java

差不多就在我的第一本書《*Using C++*》（Osborne/McGraw-Hill，1989）面世的同時，我開始教授語言。程式語言的教學已經變成了我的職業。1989 年起，我在世界各地，看到了昏昏欲睡的聽眾，有人帶著一張面無表情的臉孔，困惑的神情兼而有之。當我開始對小型團體進行內部訓練時，我在訓練過程中發掘到某些事實：即便是對我點頭微笑的學生，同樣困惑於許多議題。我發現，多年來在「軟體發展討論會（Software Development Conference）」上主持 C++（後來變成 Java）專題的工作中，我和其他講師一樣，潛意識裡想要在極短時間內告訴我的聽眾許多許多東西。由於聽眾的程度各有不同，也由於我的教材呈現方式，最終無法顧及某部份聽眾。或許如此要求有些過份，但因爲我向來反對傳統授課方式（而且我相信對大多數人們來說，反對的理由是來自於厭倦），所以我試著讓每個人都加速前進。

一度，我以十分簡短的方式做了許多不同風格的演出。最後，我結束了實驗和迭代（iteration，一種在 Java 程式設計中也會用到的技巧）」的學習歷程。我根據授課經驗所學來的事物（它讓我可以長時間快樂教學），發展出一套課程。這套課程採用分離並極易融會貫通的數個步驟來處理學習上的問題，並採取動手實踐的研討形式（這是最理想的學習方式）。在其

中，我為每一小部份課程內容設計了多個練習題。現在我將此一課程置於開放的 Java 研討課程內，你可以在 *www.BruceEckel.com* 網站上找到這份研討課程的內容。（這個研討課程的簡介也可以在書附光碟中取得。上述網站也可以找到相關資訊）

我從各個研討班獲得許多回饋訊息。在我認為我的這份課程材料足以成為一份正式的教學工具之前，這些回饋訊息對於我的課程材料的更動和調整，助益良多。儘管如此，本書絕不能以一般的研討課程筆記視之 – 我試著在這些書頁中放入儘可能多的資訊，並將這些資訊加以結構化，藉以引領你平順地前進至下一個討論主題。最重要的是，本書是為那些孤單面對一個全新語言的讀者而設計。

目標

就像我的前一本書《*Thinking in C++*》一樣，這本書結構性地環繞著程式語言教學引導上的過程。我的原始動機是希望創造一些材料，使我可以將自己在研討課程中採用的風格，融於程式語言教學之中。當我安排本書章節時，我的思考方式就像思考「如何在研討班上一堂好課」一樣。我的目標是，切割出可以在合理時數中教完而又容易被吸收的材料，並附帶適合於教室內完成的習題。

以下是本書的目標：

1. 一次呈現一個階段的題材，讓你可以在移至下一課題之前，輕鬆消化每個觀念。

2. 範例儘可能簡短、單純。但是這麼一來便會在某種程度上遠離了真實世界的問題處理方式。儘管如此，我發現，對初學者而言，詳盡理解每個範例，所帶來的愉悅勝過於了解它所能解決的問題範圍。能夠在教室中吸引學習者興趣的程式碼，數量十分有限，因此我無疑得承受諸如「使用玩具般的小例子」的種種批判，但我還是滿心歡喜地接受任何可以為教學帶來益處的形式。

3. 謹慎安排諸般特性的呈現順序，讓你不致於突兀地碰觸到任何未曾見過的內容。萬一在某些情形下無法達到此一理想，我會提出一些簡要的引導。

4. 只給你那些「我認爲對你了解此一程式語言而言十分重要」的內容，而非將我所知道的一切都倒給你。我相信「資訊重要性的階層順序」的確存在，某些題材對百分九十五的程式員來說或許沒有必要，這些資訊往往只會混淆他們的觀念，並加深他們對此語言的複雜感受而已。舉個 C 語言的例子好了，如果你能清楚記得運算子優先序（operator precedence），撰寫出精巧的程式碼想必輕而易舉。但這麼做卻有可能造成讀者、維護者的困惑。所以，忘了運算子優先序吧！在任何可能混淆的地方使用小括號不就得了。

5. 讓每一節內容有足夠的焦點，縮短授課和練習時段之間的空檔。這麼做不僅爲了讓聽眾的心態更爲主動，融入「自己動手做」的研討氣氛，而且也讓讀者更具成就感。

6. 讓你始終踏在紮實的基礎上，透過對各課題的充份了解，面對更困難的作業和書籍。

線上說明文件

Online documentation

從 Sun Microsystems 取得的 Java 程式語言及其程式庫（可免費下載），皆附有電子文件，使用 Web 瀏覽器即可閱讀。幾乎所有 Java 編譯器廠商也都提供了此份文件，或是等價的文件系統。大部份 Java 書籍也都提供此份文件的複製品。所以，除非必要，本書不會重述其內容，因爲對你而言，使用 Web 瀏覽器來找尋 classes 的說明，比遍尋全書來得快多了（更何況線上說明文件的版本可能更新）。只有當有必要補充線上文件之不足，使你得以理解特定範例時，本書才會提供 classes 的各種額外描述。

章節組織

設計本書時，有一個念頭長在我心：人們面對 Java 語言的學習路線。研討班學員所給的回饋訊息幫助我了解哪些困難部份需要詳加闡述。在那些「太過燥進、一次含入太多主題」之處，我也透過教材的呈現而一一明白：如果企圖包含許多新主題，就得全部說明清楚，因爲新東西很容易造成學生的困惑。基於這個原因，我儘可能每次只講一小段主題，避免生出太多困擾。

因此，我的目標便是每一章只教授單一主題，或是一小組相關主題，儘量不和其他主題有所牽扯。如此你便得以在前進至下一主題前，完全消化目前知識背景中的所有題材。

以下就是本書各章的簡要描述，它們分別對應於我的研討班各授課時程和練習時段。

第 1 章： 物件導論（*Introduction to Objects*）

本章提供物件導向程式編寫（object oriented programming）的概論性介紹，包括最基本問題的回答，例如物件（object）是什麼、介面（interface）與實作（implementation）、抽象（abstraction）與封裝（encapsulation）、訊息（messages）與函式（functions）、繼承（inheritance）與合成（composition），以及最重要的多型（polymorphism）。你也可以概要了解物件生成時的諸般問題，像是建構式（constructors）、物件的生命、物件被建立後置於何處、神奇的垃圾回收器（garbage collector，當物件不再被使用時能夠加以清理回收）。其他相關議題也有討論，包括異常處理（exception handling）、多執行緒（multithreading）、網絡、Internet 等等。你會看到是什麼東西使得 Java 卓越出眾，是什麼原因使得 Java 如此成功，你也會學到物件導向的分析概念和設計概念。

第 2 章： 萬事萬物皆物件（*Everything is an Object*）

本章引領你開始撰寫第一個 Java 程式。因此所有最基本的概觀都必須在這裡教給你，包括：object reference（物件引用）

觀念、物件產生方式、基本型別（primitive types）與陣列（arrays）、物件的生存空間（scoping），物件被垃圾回收器回收的方式、如何在 Java 中讓每樣東西成爲新的資料型別（class）、如何建立你自己的 classes。此外還包括：函式、引數、回傳值、名稱可視性、自其他程式庫取用組件（components）的方式、關鍵字 **static**、註解、內嵌文件。

第 3 章： 控制程式流程（*Controlling Program Flow*）

本章首先討論的是 Java 引自 C/C++的所有運算子（operators）。此外你會看到運算子的共通缺點、轉型（casting）、型別晉升（promotion）、運算優先序。然後介紹基本流程控制，這是幾乎每一個程式語言都會提供的機制：if-else 分支選擇、for/while 迴圈結構、以 break 和 continue 跳脫迴圈；其中當然包括了 Java 的標註式（labeled）break 和標註式 continue（這是爲了 Java 不再提供 goto 而設計），以及 switch/case 選擇動作。雖然大部份特性和 C/C++相同，但亦有些許差異存在。此外，所有範例皆以 Java 撰寫，因此你可以清楚看到 Java 的程式風格。

第 4 章： 初始化與清理（*Initialization & Cleanup*）

本章一開始先導入建構式（constructor）的概念，它用來保證初始化動作的順利進行。建構式的定義會牽扯到函式重載（function overloading）觀念（因爲你可能同時需要多個建構式）。接著便是清理過程的討論，這個過程的字面意義總是比實際簡單得多。正常情形下當你用完一個物件，什麼也不必操心，垃圾回收器最終會執行任務，釋放該物件所配置的記憶體。我會探討垃圾回收器及其本質。最後我們近距離觀察自動化的成員初值設定、自定的成員初值設定、初始化順序、static（靜態）初始化、以及 array 的初始化。這些細微的物件初始化動作的探討，爲本章劃上美好句點。

第 **5** 章： 隱藏實作細節（*Hiding the Implementation*）

本章討論程式碼的封裝，並解釋爲什麼程式庫中某些部份被公諸於外，某些部份卻被隱藏起來。一開始我們先檢視兩個關鍵字：**package** 和 **import**，它們在檔案層次上進行封裝，並允許你打造 classes libraries。接著探討目錄路徑和檔案名稱的問題。最後檢視 **public**、**private**、**protected** 等數個關鍵字，並引介 "friendly"（友善的）存取動作，以及不同情境下所使用的各種存取權限的意義。

第 **6** 章： 重複運用 **classes**（*Reusing Classes*）

繼承（Inheritance）幾乎是所有物件導向程式語言的標準配備。它讓我們得以使用既有的 classes，並得爲它加入額外功能（而且還可以改變，這是第七章的主題）。繼承通常用於程式碼的重複運用（reuse）：讓 base class 保持不變，只補上你想要的東西。不過，要想藉由既有的 classes 來製造新的 class，繼承並非唯一途徑。你也可以運用所謂的「複合」（composition），將物件嵌入新的 class 內。你可以在本章學到如何以 Java 的方式達到重複運用程式碼的目的，並學會如何應用。

第 **7** 章： 多型（*Polymorphism*）

多型，物件導向程式設計的基石。如果只靠自我修練，你或許得花九個月的時間才能夠體會其奧秘。透過小而簡單的範例，你將看到如何以繼承機制建立一整個族系的型別，並透過共同的 base class，操作同一族系中的物件。Java 的多型機制允許你以一般性的角度看待同一族系的所有物件。這意謂大部份程式碼不至於過度倚賴特定型別資訊，於是程式更易於延伸，並使程式的發展與原始碼的維護更加簡單，更不費力。

第 **8** 章： 介面（*Interfaces*）與內隱類別（*Inner Classes*）

Java 提供「建立重複運用關係」的第三種形式：interface，那是物件對外介面的一個純粹抽象描述。**interface** 不僅將抽象類別（abstract class）發揮到極致，由於它允許你建立某種 class，可向上轉型（upcast）至多個 base classes，所以它提供了類似 C++「多重繼承（multiple inheritance）」的變形。

inner classes（內隱類別）看起來似乎是個簡單的程式碼隱藏機制：只不過是將 class 置於另一個 class 之內而已。但其實不僅於此，它知曉 surrounding class（外圍類別）並可與之溝通。雖然 inner classes 對大多數人而言是一個全新觀念，需要花上一些時間才能無礙地利用它從事設計，但 inner classes 的確可以讓程式碼更優雅、更澄淨。

第 9 章： 物件的持有（*Holding your Objects*）

一個程式如果只是擁有帶著已知壽命的固定數量的物件，這種程式其實是相當簡單的。一般來說，你的程式會在不同時間點產生新物件，而這些時間點只有程式執行之際才能確定。此外，除非處於執行期，否則你可能無法知道你所需要的物件數量，及其確切型別。爲了解決一般化的程式問題，你必須有能力在任何時間、任意地點，產生任意數量的物件。本章深入探討 Java 2 所提供的容器程式庫（container library），讓你得以妥善保存你所需要的物件。我的討論包括最簡單的 arrays（陣列），以及 **ArrayList**、**HashMap** 之類的複雜容器（或說資料結構）。

第 10 章： 透過「異常」處理錯誤（*Error Handling with Exceptions*）

Java 的基本設計哲學是，不允許「會造成損害」的程式碼執行起來。編譯器會儘可能捕捉（catches）它所能捕捉的錯誤，但有時候某些問題 – 不論是程式員引起或程式正常執行下自然發生 – 只能夠在執行期被偵測、被處理。Java 具備了所謂的異常處理機制（exception handling），可用來處理程式執行期引發的種種問題。本章討論了 **try**、**catch**、**throw**、**throws**、**finally** 等關鍵字在 Java 中的運作方式，並說明何時才是擲出（throw）異常的最佳時機，告訴你捕捉到異常時該如何處理。此外你也會看到 Java 的標準異常，學習如何建立自定異常，並知道在建構式（constructors）中觸發異常時會發生什麼事，以及異常處理程序（exception handlers）的放置方式。

第 11 章： **Java 与 I/O 系統**（*The Java I/O System*）

理論上你可以將程式劃分爲三部份：輸入（input）、處理（process）和輸出（output）。這意謂 I/O（輸入和輸出）在此一方程式中佔了極大比重。本章可以讓你學到 Java 的諸般類別，讓你進行檔案、記憶體區塊、主控台（console）的資料讀取和寫入。本章也會提及介於「傳統 I/O」和「全新 Java I/O」之間的差異。此外本章也會檢視「將物件串流化（streaming）」（俾使物件得以被置於磁碟，或於網絡上傳遞）的過程，並討論如何將其重構（reconstructing）。這些都是透過 Java 的「物件次第讀寫」（*object serialization*）機制達成。當然，用於 Java 保存檔（**J**ava **AR**chive，JAR）格式的 Java 壓縮類別庫，也會在本章介紹。

第 12 章： **執行期型別辨識**（*Run-Time Type Identification*）

當你僅有某物件的基礎類別的 reference 時，Java 的執行期型別辨識（RTTI）機制可讓你找出該物件的確切型別。通常你應該會想要刻意忽略物件的確切型別，讓 Java 的動態繫結（dynamic binding）機制（亦即多型，polymorphism）負責展現特定型別應有的特定行爲。不過有時候，當你僅有某物件的基礎類別的 reference，而能進一步知道該物件的確切型別，可帶來很大用處。通常此一資訊讓你得以更高效率地執行某些特定動作。本章說明：（1）何謂 RTTI，（2）RTTI 的使用方式，（3）不該使用 RTTI 時，應如何避免使用。本章也介紹了 Java 的 *reflection* 機制。

第 13 章： **建立視窗和Applets**

Java 所附的 Swing GUI library，是一組可攜性高的視窗相關類別程式庫。此處所謂的視窗程式，可以是網頁內嵌小程式（applets），也可以是獨立的應用程式（applications）。本章內容包含 Swing 的簡介和 WWW applets 的開發。同時也介紹極重要的 JavaBeans 技術 — 它是所有快速應用軟體開發工具（Rapid-Application Development，RAD）的根本。

第 14 章： 多執行緒（*Multiple Threads*）

Java 提供了一些內建機制，使得單一程式內可同時並行多個被稱爲「執行緒（*threads*）」的子工作（但除非你的機器裝載多顆處理器，否則這種運作方式只是形式而已）。雖然任何地方都可以應用執行緒，但最主要還是應用於需要高度互動能力的使用者介面上。舉個例子，雖然還有一些處理動作正在進行，但使用者可以不受阻礙地按下按鈕或輸入資料。你將在本章看到 Java 多執行緒的語法和語義。

第 15 章： 分佈式計算（*Distributed Computing*）

當你開始想要撰寫運行於網絡上的程式時，一時間好像所有的 Java 特性和類別庫都一起湧現。本章探討網絡及 Internet 的通訊問題，以及 Java 提供的相關 classes。本章也介紹了重要的 Servlets 和 JSPs 觀念（兩者皆用於伺服端程式設計），以及 Java 資料庫連結機制（*Java Database Connectivity*，JDBC）、遠端函式調用（*Remote Method Invocation*，RMI）等技術。最後還介紹了 *JINI*、*JavaSpaces*、*Enterprise JavaBeans*（EJBs）等的最新技術。

附錄 A: 物件的傳遞和回傳（*Passing & Returning Objects*）

Java 允許你和物件溝通的唯一方式是，透過 reference 達成。所以，將物件傳入函式內以及函式將物件回傳，便存在著某些有趣的結果。這份附錄說明，當你將物件移入函式，或將物件自函式移出時需要知道哪些事情，才能妥善管理這些物件。本附錄也爲你介紹 **String** class 如何使用另一種截然不同的手法來解決問題。

附錄 B: **Java** 原生介面（*The Java Native Interface, JNI*）

全然可攜的 Java 程式有某些致命缺點：執行速度慢、無法存取特定平台上的服務。一旦你確切知道你的程式的執行平台，

大幅提升某些動作的執行速度是極有可能的 — 只要透過所謂的原生函式（*native methods*）即可。原生函式係以另一種語言（目前僅支援 C/C++）寫成。這份附錄爲你提供足夠的入門引導，讓你能夠寫出「和 non-Java 程式相接連」的簡單程式。

附錄 C: **Java 程式設計守則**（*Java Programming Guidelines*）
本附錄提供許多建議，幫助你進行較低層次的設計和撰碼工作。

附錄 D: 推薦讀物（*Recommended Reading*）
本附錄列出我所知道的 Java 書籍中格外有用的名單。

習題

研討班的經驗使我察覺，簡單的練習對於學生所需要的完整理解有著超乎尋常的效果。因此每章末尾我都安排了一些習題。

大多數習題都夠簡單，使你得以在指導者從旁協助的情況下，以合理的時間完成。這可確保所有學生都順利吸收了教材內容。某些習題難度較高，以免經驗豐富的學生心生厭倦。多數題目都可以在短時間內解決，並可用來檢驗所學以及鍛練自己。某些題目具有挑戰性，但其中並沒有難度很高者 — 我想你應該會自己找到這樣的題目，或者很有可能它們會自動找上門來。

某些經過挑選的習題有電子檔解答，收錄於《*The Thinking in Java Annotated Solution Guide*》，僅須小額付費即可自 *www.BruceEckel.com* 下載。

多媒體光碟（Multimedia CD ROM）

本書有兩張相關的多媒體光碟。第一張光碟《*Thinking in C*》隨書附贈。本書前言最末曾對此光碟有些介紹。CD 之中準備了一些相關材料，讓你可以加速學習必要的 C 語法 — 這是學習 Java 不可或缺的一步。

第二張多媒體光碟也和本書內容有關。這是一份獨立產品，其中含有一週的「Java 動手做（Hands-On Java）」訓練課程的所有內容。我所錄的講課內容長度超過十五小時，並整合上百張投影片。由於我的研討班課程係以本書為基礎，所以這是極為理想的補充教材。

這張光碟還含有五天時程的精修班課程內容（其主題將和個人著重方向有密切的關係）。我們相信，它為品質樹立了新的標準。

如果你需要「Java 動手做」光碟，請向 *www.BruceEckel.com* 網站訂購。

原始碼（Source code）

本書所有原始碼都被我宣告為自由軟體（freeware），以單一包裝形式進行傳佈。只需訪問 *www.BruceEckel.com* 網站即可取得。如果想要確認你所拿到的是否為最新版本，上述網站是此一產品的官方站台。或許你可以從其他網站取得這份產品的鏡像版本（mirrored version），不過你應該到官方網站上確認，確保你所拿到的鏡像產品的確是最新版。你有權力在你的課程或其他教育場合傳佈這些程式碼。

以下版權宣告的主要目的，是希望確保原始碼皆被適當引用，並且不希望你在沒有獲得允許的情況下，自行透過印刷媒體重新發行這些程式碼。只要你引用了以下聲明，那麼一般而言在大部份媒體上使用本書範例都不會帶給你任何麻煩。

你可以在每一個原始碼檔案中看到如下的版權宣告：

```
//:! :CopyRight.txt
Copyright ©2000 Bruce Eckel
Source code file from the 2nd edition of the book
"Thinking in Java." All rights reserved EXCEPT as
allowed by the following statements:
You can freely use this file
```

```
for your own work (personal or commercial),
including modifications and distribution in
executable form only. Permission is granted to use
this file in classroom situations, including its
use in presentation materials, as long as the book
"Thinking in Java" is cited as the source.
Except in classroom situations, you cannot copy
and distribute this code; instead, the sole
distribution point is http://www.BruceEckel.com
(and official mirror sites) where it is
freely available. You cannot remove this
copyright and notice. You cannot distribute
modified versions of the source code in this
package. You cannot use this file in printed
media without the express permission of the
author. Bruce Eckel makes no representation about
the suitability of this software for any purpose.
It is provided "as is" without express or implied
warranty of any kind, including any implied
warranty of merchantability, fitness for a
particular purpose or non-infringement. The entire
risk as to the quality and performance of the
software is with you. Bruce Eckel and the
publisher shall not be liable for any damages
suffered by you or any third party as a result of
using or distributing software. In no event will
Bruce Eckel or the publisher be liable for any
lost revenue, profit, or data, or for direct,
indirect, special, consequential, incidental, or
punitive damages, however caused and regardless of
the theory of liability, arising out of the use of
or inability to use software, even if Bruce Eckel
and the publisher have been advised of the
possibility of such damages. Should the software
prove defective, you assume the cost of all
necessary servicing, repair, or correction. If you
think you've found an error, please submit the
correction using the form you will find at
www.BruceEckel.com. (Please use the same
form for non-code errors found in the book.)
///:~
```

只要每個原始碼檔案依舊保有上述版權宣告，你便可以在自己的專案中或課堂上，或是你的簡報材料中，使用這些程式碼。

撰碼標準（Coding standards）

本書內文中，識別字（identifiers，包括函式、變數、類別等名稱）皆以粗體表示。大多數關鍵字亦以粗體表示，但是頻繁出現的關鍵字（例如 class）就不如此，以免令人生厭。

我在本書範例中採用特定的撰碼風格（coding style）。此風格依循 Sun 在（幾乎）所有程式碼中的風格 — 你可以從其網站（*java.sun.com/docs/codeconv/index.html*）上找到。此風格似乎也被大多數 Java 開發環境支援。如果你已讀過我的其他著作，你可能會注意到，Sun 的撰碼風格和我一致 – 這讓我滿心歡喜，雖然我沒有爲此做任何事。至於程式碼格式（formatting style），這個主題適合花好幾個鐘頭來辯論，所以我不打算透過我的作法來規範所謂的正確性；我對於我所採用的風格，有一些自己的想法。由於 Java 是一種自由格式（free-form）的語言，所以你大可使用任何讓你感到自在的風格，無妨。

本書程式碼，都直接來自編譯過的檔案，透過文字處理器，以文字形式呈現。因此這些程式碼應該都能正常運作，不致於出現編譯錯誤。所有會導致編譯錯誤的錯誤，皆以註解符號（//!）標記起來。因此你可以輕易發現它們，並以自動化方式檢測它們。如果你發現程式碼有錯並回報給作者，我會把你的大名列於將被傳佈出去的原始程式碼中，並列名於再版書冊中以及 *www.BruceEckel.com* 網站上。

Java 版本

當我要判斷某一行程式碼是否正確時，我通常依據 Sun 的 Java 產品（編譯器）來裁量。

Sun 依序推出了三個 Java 主要版本：1.0, 1.1, 2。關於最後一項，雖然 Sun 推出的 JDK 仍然採用 1.2, 1.3, 1.4 的編號，但一般都統稱爲版本 2）。版本

2 似乎將 Java 帶上了黃金時期，尤其是使用者介面工具被格外重視的此刻。本書集中火力探討 Java 2，並以它來進行測試 – 雖然有時候我得做些讓步，俾使程式碼得以在 Linux 上編譯 — 透過從 Linux 取得的 JDK。

如果你曾學習過 Java 語言的較早版本，而其內容未被本書涵蓋，那麼，本書第一版仍可自 *www.BruceEckel.com* 免費下載，並且也附於本書光碟。

還有一件事請注意，當我需要提及較早版本時，我不會提及次修正版號。本書之中我只會採用 Java 1.0, Java 1.1, Java 2 等版本號碼。

研討課程與顧問指導

我的公司提供五天時程的公開訓練課程，以親手實踐的形式進行，依據的材料即為本書。課堂所授內容，是書中每一個章節挑選出來的材料。每堂課之後都有指導時段和練習時段，讓每位學員都能夠得到個別的照料。簡介性的課程，其錄音教材和投影片都已置於書附光碟中，你不需要長途跋涉，也不需要花費金錢，就可以獲得些許研討經驗。如果想獲得更多資訊，請上 *www.BruceEckel.com* 網站。

我的公司也提供諮詢、顧問指導、演練服務，藉以引導你的專案計畫順利走過開發週期 – 特別是面對公司的第一個 Java 專案時。

關於錯誤

不論作者有多少秘訣可以找出錯誤，總是會有一些錯誤蔓延出來，而且對讀者造成困擾。

本書的 HTML 版（可自書附光碟取得，也可自 *www.BruceEckel.com* 下載）的每一章啓始處，都有一個鏈結，可連結到「錯誤提報功能」。當然，網站上的本書相關網頁也有這個功能。如果你發現任何錯誤，請使用此一表單提報給我，並請附上你的修正建議。如有必要，也請附上原本的原始碼，並標註你的任何修正建議。我將對你的幫助致以無上的感激。

封面故事

《*Thinking in Java*》的封面靈感，來自於美國的 Arts & Crafts 運動。這個運動約略始於二十世紀初，並於 1900~1920 達到巔峰。它發源於英格蘭，是對工業革命所帶來的機器生產及維多利亞時期繁複裝飾風格的排拒。Arts & Crafts 強調簡約設計、自然形式、純手工打造、以及獨立工匠的重要性。它極力避免使用現代化工具。這和今天我們所面對的種種情境有著許多類似：世紀之交、從「電腦革命的濫觴」到「對個人更完備更具意義」的演化過程、對軟體技巧（而非只是製造程式碼）的強調。

我以同樣的態度來看待 Java：一種力量，試圖將程式員從作業系統的技工層次提升出來，朝向「軟體工藝師」的目標前進。

本書作者和封面設計者是童年好友，我們都從這個運動中獲得靈感，也都擁有各種起源於此一時期（或受其啓發）的各種家具、燈具、種種器物。

本書封面的另一個用意代表著，博物學家所可能展示的昆蟲標本收集盒。這些昆蟲本身都是物件，都被置於「盒子」這樣的物件中。盒子又被置於「封面」這樣的物件中。這說明了物件導向程式設計極爲基礎的「集成（*aggregation*）」概念。當然，程式員可能從中得不到任何助益，卻聯想到所謂的程式臭蟲（bugs）。這些臭蟲被捕捉，然後在樣本罐中被殺掉，最後被固定於小小的展示盒中。這或許可以類比爲：Java 有能力搜尋、顯示、制服臭蟲。這也是 Java 極具威力之眾多特性中的一個。

致謝（Acknowledgements）

我首先要感謝所有和我一起教授課程、一起進行諮詢、一起發展教學計畫的夥伴們：Andrea Provaglio、Dave Bartlett（他對第 15 章有卓越的貢獻）、Bill Venners、Larry O'Brien。當我嘗試持續爲那些和我們一樣團隊工作的其他群眾們發展最佳模式時，你們所展現的耐心使我銘感五內。我也要謝謝 Rolf Andre Klaedtke（瑞士）；Martin Vlcek、Martin Byer、Vlada、Pavel Lahoda、Martin the Bear 以及 Hanka（布拉格）；還有 Marco Cantu（義大利），他在我第一次策劃歐洲研討課程時，和我共同達成了任務。

我也要謝謝 Doyle Street Cohousing Community 在我撰寫本書第一版的兩年內，對我多有容忍。非常感謝 Kevin 和 Sonda Donovan，在我撰寫本書第一版的暑假期間，將他們最豪華的 Crested Butte 租給我使用。我同時要感謝 Crested Butte 及 Rocky Mountain Biological Laboratory 的眾多友善居民們，讓我有賓至如歸的感覺。我還要謝謝 Moore Literacy Agency 的 Claudette Moore，因爲她的無比耐性和堅忍不拔的毅力，我才得到了我想要的完美效果。

我的前兩本書在 Prentice-Hall 出版時，Jeff Pepper 是我的編輯。Jeff 總是在正確的時機和正確的地點出現，清除所有障礙，並做了所有正確的事，成就此次極爲愉悅的出版經驗。謝謝 Jeff，這對我來說意義深遠。

我還要特別感謝 Gen Kiyooka，以及他的公司 Digigami。前幾年我置放各種素材所需的 Web 伺服器，都由他熱心提供。這是無價的援助。

我也要謝謝 Cay Horstmann（《*Core Java*》作者之一，Prentice-Hall, 2000）、D'Arcy Smith（Symantec）、Paul Tyma（《*Java Primer Plus*》作者之一，The Waite Group, 1996），他們對我在 Java 語言觀念上的釐清，提供了莫大的幫助。

我也要謝謝那些曾經在軟體開發研討會（Software Development Conference）上由我主持的 Java 專題上發言的人們，以及那些在研討班上提問，使我得以參考並使教材更清楚的學生們。

特別感謝 Larry 和 Tian O'Brien，你們將我的研討課程轉製爲《*Hands-On Java*》光碟。

將修正意見回饋給我的好心人們，你們的幫助使我受惠匪淺。第一版特別要感謝的是：Kevin Raulerson（找出了成堆的臭蟲）、Bob Resendes（簡直太了不起了）、John Pinto、Joe Dante、Joe Sharp（三位都優秀得令人難以置信）、David Combs（訂正了許多文法和說明）、Dr. Robert Stephenson、John Cook、Franklin Chen、Zev Griner、David Karr、Leander A. Stroschein、Steve Clark、Charles A. Lee、Austin Maher、Dennis P. Roth、Roque Oliveira、Douglas Dunn、Dejan Ristic、Neil Galarneau、David B. Malkovsky、Steve Wilkinson、還有許許多多人，此處實難備載。本書第一版在歐洲發行時，Ir. Marc Meurrens 教授在電子版的宣傳與製作上做了十分卓絕的努力。

在我生命中，有許多技術出眾的人們變成了我的朋友。在他們所做的瑜珈及其他形式的精神鍛練上，我得到了十分特別的靈感與引導。這些朋友是 Kraig Brockschmidt、Gen Kiyooka、Andrea Provaglio（在義大利，他協助我以更一般化的角度來理解 Java 和程式設計。現在的他是美國 MindView 團隊中的一員）。

對 Delphi 的理解也幫助我認識 Java，這一點不值得驚訝。因爲語言設計上的許多概念和決定都是共通的。我的 Delphi 朋友們協助我洞察神秘的程式開發環境。這些朋友是 Marco Cantu（另一位義大利朋友，或許正沉浸在拉丁語帶來的程式語言靈光之中）、Neil Rubenking（他在發現電腦的奧妙之前，經歷過瑜珈、素食、禪修），當然還有 Zack Urlocker，和我一同旅行世界的長期夥伴。

我的朋友 Richard Hale Shaw 的卓越洞見和支援，都帶給我無上的幫助（Kim 也是）。Richard 和我花了數個月的時光一起教授研討課，一起試著找出對聽眾而言最完美的學習經驗。在此我也要謝謝 KoAnn Vikoren、

Eric Faurot、Marco Pardi、以及 MFI 的其他工作夥伴。格外謝謝 Tara Arrowood，重新啓發了我對研討會的可行性想法。

本書設計、封面設計、封面圖像，皆出自於我的朋友 Daniel Will-Harris，他是一位相當著名的作家和設計家（*www.Will-Harris.com*）。在電腦和桌上出版發明之前，他就已經在國中時期玩過所謂的 rub-on letters，並且抱怨我的代數含糊不清。不過我現在已經能夠自己完成出版頁稿了，所以排版上的所有問題都應該算我頭上。我使用 Microsoft Word 97 for Windows 撰寫本書，並使用 Adobe Acrobat 製作出版頁稿；本書直接以 Acrobat PDF 檔案製作。我兩次在海外完成本書定稿，第一版在南非開普敦，第二版在布拉格。這是電子時代的明證。我所使用的主要字體是 *Georgia*，標題採用 *Verdana*。封面字體是 *ITC Rennie Mackintosh*。

我也要對製造編譯器的廠商致上謝意：Borland、Linux 的 Blackdown 團隊、以及絕對不能不提的 Sun。

我的所有老師、所有學生（也可視爲我的老師）都應該接受我的特別謝意。最有趣的寫作老師是 Gabrielle Rico（《*Writing the Natural Way*》一書作者，Putnam, 1983）。我對於 Esalen 所發生的那了不起的一週永銘於心。

提供支援的朋友們還包括（恐有遺漏）：Andrew Binstock、Steve Sinofsky、JD Hildebrandt、Tom Keffer、Brian McElhinney、Brinkley Barr、Bill Gates（Midnight Engineering Magazine）、Larry Constantine 和 Lucy Lockwood、Greg Perry、Dan Putterman、Christi Westphal、Gene Wang、Dave Mayer、David Intersimone、Andrea Rosenfield、Claire Sawyers、以及眾多的義大利朋友（Laura Fallai、Corrado、Ilsa、Cristina Giustozzi）、Chris 和 Laura Strand、the Almquists、Brad Jerbic、Marilyn Cvitanic、the Mabrys、the Haflingers、the Pollocks、Peter Vinci、the Robbins Families、the Moelter Families（和 the McMillans）、Michael Wilk、Dave Stoner、Laurie Adams、the Cranstons、Larry Fogg、Mike 和 Karen Sequeira、Gary Entsminger 和 Allison Brody、Kevin Donovan 和 Sonda Eastlack、Chester 和 Shannon

Andersen、Joe Lordi、Dave 和 Brenda Bartlett、David Lee、the Rentschlers、the Sudeks、Dick、Patty、Lee Eckel、Lynn 和 Todd 和其家族成員。當然，還有我摯愛的爸媽。

Internet 上的貢獻者

謝謝這些助我改用 Swing 程式庫撰寫範例並提供其他協助的人們：Jon Shvarts、Thomas Kirsch、Rahim Adatia、Rajesh Jain、Ravi Manthena、Banu Rajamani、Jens Brandt、Nitin Shivaram、Malcolm Davis，以及所有曾經提供協助的人們。你們真的幫助我開展了這個計畫。

1: 物件導論

Introduction to Objects

電腦革命始於機器。因此，程式語言的發軔也始於對機器的模仿。

不過，電腦並非是那麼冷冰冰的機器。電腦是意念發揮的工具（一如 Steve Jobs 常喜歡說的「意念的自行車」一樣），並且也是一種不同類型的表達媒介。這個工具愈偏離機器的長相，就愈像我們頭腦的一部份，一如寫作、繪畫、雕刻、動畫、電影等意念表達形式。物件導向程式設計（Object-oriented Programming，OOP），便是這樣一個以電腦做爲表達媒介的巨大浪潮中的一環。

本章將爲你介紹基本的 OOP 觀念，並涵括軟體開發方法的概論性介紹。本章，甚至整本書，都假設你對程序性語言（procedural programming language）有著某種程度的經驗，我所謂程序性語言不一定得是 C。如果你覺得有必要在接觸此書之前先在程式設計和 C 語法上多下功夫，你可以研讀本書所附的培訓光碟《*Thinking in C: Foundations for C++*》，其內容也可以從 *www.BruceEckel.com* 取得。

本章提供的是背景性、補充性的材料。許多人在沒有看清整個物件導向程式設計方法的完整面貌之前，無法自在地從事此類設計活動。因此，我將引入許多觀念，爲你奠定 OOP 的紮實基礎。另外還有許多人在沒有看到某種程度的實際運作機制之前，無法看清物件導向程式設計方法的完整面貌。這樣的人如果沒有程式碼在手，很容易迷失方向。如果你正是這種人，而且渴望早點知道 Java 語言的細節，請你從容跳過本章，這並不會影響你的程式撰寫和語言學習。不過，相信我，最終你還是需要回過頭來填補必要的知識，藉以了解物件的重要，以及「透過物件進行設計」的方式。

抽象化的過程

The progress of abstraction

所有程式語言都提供抽象化機制（abstraction）。甚至可以大膽地說，我們所能解決的問題的複雜度，取決於抽象化的類型和品質。我所謂類型，指的是「你所抽象化的事物爲何？」組合語言僅對底層的實體機器進行少量抽象化。許多所謂命令式（imperative）程式語言（例如 Fortran、BASIC、C），則在組合語言之上再抽象化。此類語言大幅改進了組合語言，但它們所做的主要是機器本身的抽象化，你依舊無法逃脫「以電腦結構進行問題思考」的命運，因而無法以待解問題的結構來做爲思考基準。程式設計者必須自行建立介於機器模型（位於你所建立之問題模型的解域（solution space，例如電腦）內）和實際待解問題模型（位於問題實際存在之題域（problem space）內）之間的關聯性。這裡頭需要的便是「對映（mapping）」功夫，然而這種能力並非程式語言的自性本質，這使得程式難以撰寫，維護代價高昂。於是才產生出「程式方法（programming methods）」這個產業。

另一種建立機器模型的方式，便是建立待解問題的模型。早期的程式語言如 LISP 和 APL 都選擇了觀看世界的某種特定方式，分別認爲「所有的問題最終都是 lists」、「所有的問題都是演算形式（algorithmic）」。PROLOG 則將所有問題轉換爲一連串決策（chains of decisions）。另外也有基於制約條件（constraint-based）的程式語言，以及專門處理圖形化符號的程式語言（後者已被證明束縛太多）。這些方式對於它們所瞄準的特定題型，都能提供不錯的解決方案，然而一旦跳脫特定領域，就顯得時地不宜。

物件導向法（Object Oriented approach）更進一步，提供各式各樣的工具，讓程式設計者得以在題域（problem space）中表現必要的元素。這種

表達方式具備足夠的一般化，使程式設計者不必受限於任何特定題型。我們將題域中的元素和其在解域（solution space）中的表述（representation）稱爲「物件」（當然你還需要其他一些無法被類比爲題域內的元素的物件）。這其中的觀念是，程式可以透過「導入新型物件」而讓自己得以適用於特定領域的問題；當你閱讀解法的程式碼時，便如同閱讀問題本身的表述一樣。這種語言比我們過去所擁有的任何語言具備更彈性更威力的抽象化機制。因此 OOP 提供了以問題描述問題（describe the problem in terms of the problem）的能力，而不再是以解答執行之所在（電腦）的形式來描述問題。不過，當然了，最終還是會接回電腦本身。每個物件看起來都有點像是一部微型電腦，有著自身的狀態，你也可以要求執行它所提供的種種操作（operations）。如果把它們類比至真實世界，以這種角度來說似乎不錯：它們都有特性（characteristics）和行爲（behaviors）。

有些程式語言設計者認爲，單靠物件導向程式設計本身，還不足以輕易解決所有程式設計問題，因而倡議所謂的「多模式（multiparadigm）」程式語言，試圖融合多種不同的解決方案[1]。

Alan Kay 曾經摘要整理了 Smalltalk 的五大基本特質。而 Smalltalk 正是第一個成功的物件導向程式語言，同時也是 Java 以爲根基的語言之一。Smalltalk 的特性代表物件導向程式設計最爲純淨的一面：

1. **萬事萬物皆物件**。將物件視爲神奇的變量，除了可以儲存資料之外，你還可以「要求」它執行自身所具備的操作能力。理論上你可以將待解問題中的所有觀念性組成，都變成程式中的物件。

2. **程式便是成堆的物件，彼此透過訊息的傳遞，請求其他物件進行工作**。如果想對物件發出請求（request），你必須「傳送訊息」至該物件。更具體地說你可以把訊息想像是對「隸屬某個特定物件」的函式的呼喚請求。

---

[1] 請參考 Timothy Budd，《*Multiparadigm Programming in Leda*》，Addison-Wesley，1995.

3. 每個物件都擁有由其他物件所構成的記憶。你可以藉由「封裝既有物件(s)」的方式來產生新型態的物件。因此你可以在程式中建立複雜的體系，卻將複雜的本質隱藏於物件的單純性之下。

4. 每個物件皆有其型別（type）。就像「每個物件皆爲其類別（class）的一個實體（instance）」這種說法一樣，類別（class）即型別（type）的同義詞。不同的類別之間最重要的區分就是：你究竟能夠發送什麼訊息給它？

5. 同一型別的所有物件接受的訊息皆相同。這句話在稍後還會陸續出現。由於「圓形」物件同樣也是「幾何形」物件，所以「圓形」肯定能接受所有可以發送給「幾何形」的訊息。這意謂你可以撰寫和「幾何形」溝通的程式碼，並自動處理所有與「幾何形」性質相關的事物。這種「替代能力（substitutability）」正是 OOP 中最具威力的概念之一。

每個物件都有介面

An object has an interface

亞理斯多德或許是第一個深入考究「型別（type）」的哲人；他曾提過魚類和鳥類（the class of fishes and the class of birds）這樣的字眼。史上第一個物件導向程式語言 Simula-67，則是透過基礎關鍵字 **class**，將新型別導入程式中，從而直接引用了類別的概念。這個概念是：所有物件都是獨一無二的，但也都是「屬於同一類別、有著共同特性和行爲」之所有物件的一部份。

Simula，一如其名，誕生目的是爲了發展模擬程式。例如古典的「銀行出納員問題」中存在許多出納員、客戶、帳號、交易、以及金錢單位，這正是許多「物件」的寫照。「隸屬同一類別」的眾多物件除了在程式執行期具備不同的狀態外，其餘完全相同，這也正是 **class** 這個關鍵字的由來。建立抽象資料型別（類別），是物件導向程式設計中的基本觀念。抽象資料型別的運作方式和內建基本型別幾乎沒什麼兩樣。你可以產生隸屬某個

型別的變數（以物件導向的說法，此可稱爲物件或實體，instance），也可以操作這些變數（這種行爲亦稱爲「遞送訊息」或「發出請求」；是的，你送出訊息，物件便能夠知道此訊息相應的目的）。每個類別的成員（members），或稱爲元素（elements），都共用相同的性質，例如每個帳戶都有結餘金額，每個出納員都能夠處理存款動作…等等。此外，每個成員也都有其自身狀態，例如每個帳戶都有不同的結餘金額，每個出納員都有各自的姓名。於是出納員、客戶、帳戶、交易等等在電腦中都可以被表現爲獨一無二的個體。這樣的個體便是物件，每個物件都隸屬於特定的類別，該類別定義了物件的特性和行爲。

所以，雖然我們在物件導向程式設計過程中，實際所做的是建立新的資料型別（data types），但幾乎所有物件導向程式語言都使用 class 這個關鍵字來表示 type。當你看到型別（type）一詞時，請想成是類別（class），反之亦然[2]。

由於 class 描述了「具有共同特性（資料元素）和共同行爲（功能）」的一組物件，所以 class 的的確確就是 data type，就像所有浮點數都有一組共通的特性和行爲一樣。箇中差異在於，程式設計者藉由定義 class 以適應問題，而不再被迫使用那些現成的 data types — 它們僅僅被設計用來表示機器中的某個儲存單元。你可以針對自己的需求，加入新的 data types，因而擴展程式語言的功能。程式設計系統欣然接受新的 classes，同時也賜予它們和內建 types 一樣的照料及一樣的型別檢驗（type- checking）。

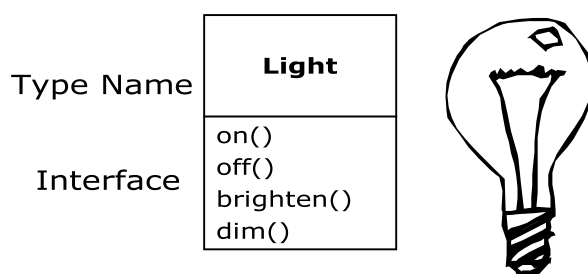
物件導向方法並不侷限於模擬程式的發展。無論你是否同意「所有程式都是用來模擬你正在設計的那個系統」這一論點，OOP 技術都可以輕易簡化許多大型問題，得到簡單的解決方案。

一旦 class 建立之後，隸屬該 class 的物件，你想要多少個就能產生多少個。你可以操作這些物件，就像它們是存在於待解問題中的元素一般。

[2] 某些人對此還是有所區別，他們認爲 type 決定了介面（interface），而 class 則是該介面的一個特定實作品。

確實，物件導向程式設計的挑戰之一，便是在題域內的眾多元素與解域內的眾多物件之間，建立起一對一的對映。

現在，問題來了，你該如何令物件爲你所用呢？必須有某種方式對物件發出請求，使物件能夠做些諸如完成一筆交易、在螢幕上繪圖、打開某個開關之類的工作。每個物件都只能滿足某些請求。物件的介面（interface）定義了它所接受的請求內容，而決定介面的便是 type（型別）。我們可以拿電燈泡做一個簡單的比喻：




```
Light lt = new Light();
lt.on();
```

介面（interface）規範了你能夠對物件發出的請求。不過，還是得有程式碼來滿足這些請求。這些程式碼加上被隱藏的資料，構成所謂的實作（implementation）。從程序式設計（procedural programming）的觀點來看，並沒有太複雜。每個 type 都有一些函式對映於任何可能收到的請求。當你對某個物件發出某個請求，某個函式便被喚起。此一過程通常被扼要地說成：你送出訊息（發出請求）至某物件，該物件便知道此一訊息的對應目的，進而執行起對應的程式碼。

本例之中的 type/class 名稱是 **Light**，特定的 **Light** 物件名爲 **lt**。你能夠對 **Light** 物件發出的請求是：將它打開、將它關閉、使它亮些、使它暗些。本例產生一個 **Light** 物件的方式是：定義 **lt** 這個物件名稱，並呼叫 **new** 請求產生該種型別的物件。欲發送訊息給物件，可以先標示出物件名稱，再以句點符號（dot）連接訊息請求。從使用者的觀點出發，這種「以物件來進行設計」的型式很漂亮。

上圖是以所謂 UML（*Unified Modeling Language*）形式呈現：每個 class 皆以矩形方格表示，class/type 名稱位於方格上方，你所關心的任何 data members（資料成員）都置於方格中央，方格下方放置所謂的 member functions（成員函式），這些函式隸屬於此一物件，能夠接收你所發送的訊息。通常只有 class 名稱及公開的（public）member functions 會被顯示於 UML 圖中，方格中央部份不繪出。如果你只在意 class 名稱，那麼甚至方格下方的部份也沒有必要繪出。

被隱藏的實作細節

The hidden implementation

將程式開發人員依各自的專業領域加以區分，對我們的概念釐清大有幫助。程式開發人員可分為：開發新資料型別的所謂 class 創造者，以及在應用程式中使用他人所開發之 classes 的所謂客端程式員（*client programmers*）[3]。客端程式員的目標是收集許多可供運用的 classes 以利快速開發應用程式。Class 創造者的目標則是打造 classes，並且只曝露客端程式員應該知道的事物，隱藏其他所有事物。為什麼？因為如果加以隱藏，客端程式員便無法使用，這意謂 class 創造者可以改變隱藏的部份，不必擔心對其他人造成衝擊。隱藏部份通常代表物件內部脆弱的一環，它們很容易被不小心或不知情的客端程式員毀壞掉。因此將實作部份隱藏起來可以減少程式臭蟲。實作隱藏（implementation hiding）的觀念再怎麼強調也不過份。

在任何相互關係中，存在一個「參與者共同遵守的界限」是一件重要的事情。當你建立一個 class library 時，你會和客端程式員建立起關係。他可能在某個程式中使用你的 library，也可能建立一個更大的 library。如果

[3] 關於這個詞彙的使用，我得感謝我的朋友 Scott Meyers。

任何人都可以取用某個 class 的所有 members，那麼客端程式員便可以對 class 做任何事情，不受任何管束。你可能希望客端程式員不要直接操作你的 class 中的某些 members，但如果缺少某種「存取權限控管機制」，就無法杜絕此事，導致每個物件都赤裸裸地攤在陽光下。

因此，「存取權限控管機制」的第一個存在理由便是，讓客端程式員無法碰觸他們不該碰觸的事物 － 這些部份應該僅供 data type 內部使用，而非被外界用來解決特定問題。這對使用者而言其實也是一種服務，因爲使用者可以輕易看出哪些事物對他們來說重要，哪些可以忽略。

「存取權限控管機制」的第二個存在理由是，讓 library 設計者得以改變 class 內部運作方式而不擔心影響客端程式。舉例來說，你可能想要簡化開發動作，改以較簡單的方式來實作某一特定 class。但稍後卻發現，你得重新寫過才能改善其執行速度。如果介面和實作二者能夠切割清楚，這個工作便輕而易舉。

Java 使用三個關鍵字來設定 class 的存取界限：**public**、**private**、**protected**。這些關鍵字的意義和用法相當直覺。這些被稱爲「存取指定詞（access specifiers）」的關鍵字，決定了誰才有資格使用其下所定義的東西。接續在 **public** 之後的所有定義，每個人都可取用。接續在 **private** 之後的所有定義，除了型別開發者可以在該型別的 member functions 中加以取用，沒有其他任何人可以存取。**private** 就像是你和客端程式員之間的一堵牆，如果有人企圖存取 **private** members，會得到編譯期錯誤訊息。**protected** 和 **private** 很相像，只不過 class 繼承者有能力存取 **protected** members，卻無法存取 **private** member。稍後還會有對繼承（inheritance）的簡短介紹。

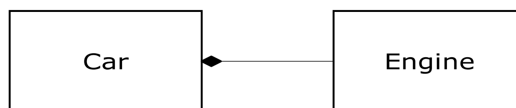
Java 還有一種所謂的「預設（default）」存取權限。當你沒有使用上述任何一個指定詞時，用的便是這種存取權限。有時候這被稱爲 friendly 存取權限，因爲同一個 package 中的其他 classes，有能力存取這種所謂的 friendly members，但在 package 之外，這些 friendly members 形同 **private** members。

重複運用實作碼

Reusing the implementation

一旦 class 開發完成並經測試，它應該（理想情形下）代表著一份有用的程式單元（unit of code）。雖然很多人都對復用性（reusability）有著熱切的期望，但事實證明，欲達此目的並不容易，你得具備豐富的經驗和深刻的見解。一旦某個 class 具備了這樣的設計，它便可以被重複運用。程式碼的重複運用，是物件導向程式設計所提供的最了不起的優點之一。

想要重複運用某個 class，最簡單的方式莫過於直接使用其所產生的物件。此外你也可以把某個 class 物件置於另一個 class 內。我們稱這種形式為「產生一個成員物件」。新的 classes 可由任意數目、任意型別的它種物件組成，這些物件可以任何組合方式達到你想要的功能。由於這種方式是「以既有的 classes 合成新的 class」，所以這種觀念被稱為「複合（*composition*）」或「聚合（*aggregation*）」。複合通常被視為 "has-a"（擁有）的關係，就好像我們說「車子擁有引擎」。



（以上 UML 圖以實心菱形指向車子，代表複合關係。我通常採用更簡單的形式，只畫一條線而不繪出菱形，來代表聯繫（association）關係[4]。

透過複合，程式員可以取得極大彈性。class 的成員物件通常宣告為 **private**，使客端程式員無法直接取用它們。這也使你得以在不干擾現有

---

[4] 在大多數示意圖中，這樣的表示便已足夠，通常你不需要在意使用的究竟是聚合或是複合。

客戶程式碼的情形下，更動這些成員。你也可以在執行期改變成員物件，藉以動態改變程式行爲。稍後即將探討的「繼承（inheritance）」關係，由於編譯器會對透過繼承而產生的 class 加上諸多編譯期限制，因此繼承不具備這樣的彈性。

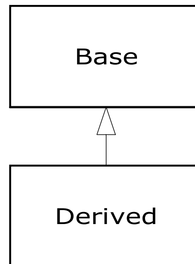
由於繼承在物件導向程式設計中如此重要，使得它常常被高度地、甚至過度地強調。程式設計新手於是會有一種刻板印象，以爲「應該處處使用繼承」。這會造成誤用，並導致過於複雜的設計。事實上在建立新 class 時，你應該先考慮複合（composition），因爲它夠簡單又具彈性。如此一來你的設計會更加清晰。有了一些經驗之後，便更能看透繼承的必要運用時機。

繼承：重複運用介面

Inheritance: reusing the interface

物件這個觀念，本身就是十分好用的工具，讓你得以透過概念（concepts）將資料和功能封裝在一起，因而表述出題域（problem space）中的想法，不必受迫於使用底層機器語言。這些概念係以關鍵字 **class** 來表現，成爲程式語言中的基本單位。

可惜的是，這樣還是有許多麻煩：建立某個 class 之後，即使另一個新的 class 有著相似功能，你還是被迫重頭建立新的 class。如果我們能夠站在既有基礎上，複製 class 的內容，然後這邊加加、那邊改改，可就真是太好了。事實上透過繼承便可達到如此的效果。不過有個例外：當原先的 class（稱爲 *base class* 或 *super class* 或 *parent class*）發生變動時，修改過的「複製品」（稱爲 *derived class* 或 *inherited class* 或 *sub class* 或 *child class*）也會同時反映這些變動。



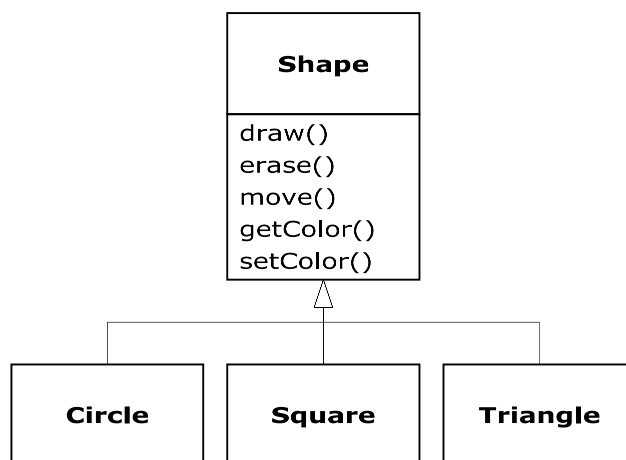
（以上 UML 圖中的箭號，是從 derived class 指向 base class。稍後你便能夠理解，可以存在一個以上的 derived classes。）

type 不僅僅只是用來描述一組物件的制約條件，同時也具備了與其他 types 之間的關係。兩個 types 可以有共通的特性和行爲，但其中某個 type 也許包含較多特性，另一個 type 也許可以處理較多訊息（或是以不同的方式來處理訊息）。所謂「繼承」便是透過 base types 和 derived types 的觀念，表達這種介於 type 和 type 之間的相似性。Base type 內含所有 derived types 共享的特性和行爲。你可以使用 base type 代表系統中某些物件的核心概念，再以 base type 爲基礎，衍生出其他 types，用來表示此一核心部份可被實現的種種不同方式。

以垃圾回收機（trash-recycling machine）爲例，它用來整理散落的垃圾。假設 base type 是「垃圾」，那麼每一袋垃圾都有重量、價值等特性，可被切成絲狀、可被熔化或分解。以此爲基礎，可以衍生出更特殊的垃圾型式，具備額外的特性（例如罐子可以有顏色）或行爲（鋁罐可壓碎、鐵罐具有磁性）。此外，它們的某些行爲可能不同（例如紙張的價值便和其種類與狀態有關）。透過繼承的使用，你可以建立一個型別階層體系（type hierarchy），表現出你想要解決的問題。

第二個例子是經典的 shape（幾何形狀）範例，可能用於電腦輔助設計系統或模擬遊戲之中。Base type 便是 "shape"，擁有大小、顏色、位置等特性，並且可被繪製、擦拭、移動、著色。以此爲基礎，便可衍生出各種特定的幾何形狀出來：圓形、正方形、三角形…，每種形狀都可以擁有額外的特性和行爲，例如某些形狀可以被翻轉。某些行爲也許並不相同，例如

面積計算的方式便不盡相同。型別階層體系（type hierarchy）同時展現了各種形狀之間的相似性和相異性。

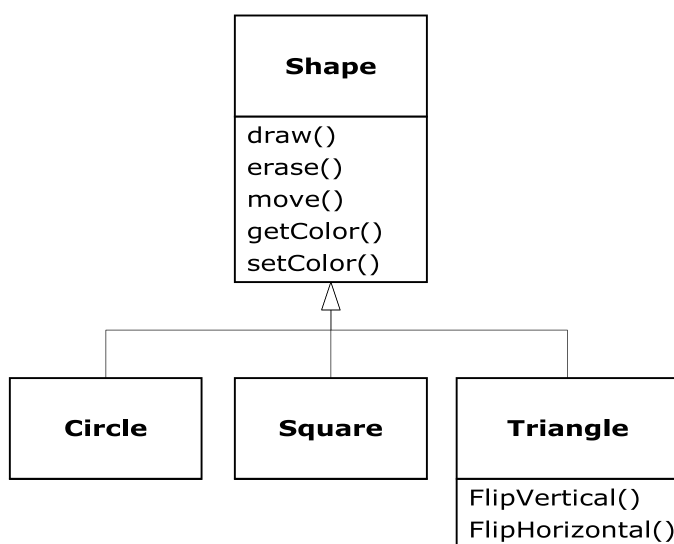


如果我們能以問題原本所用的術語來轉換解答，將會大有益處，因爲你不需要在問題的描述和解答的描述之間，建立起眾多中介模型。透過物件的使用，type hierarchy（型別階層體系）成了主要模型，讓你得以直接自真實世界出發，以程式碼來描述整個系統。是的，對使用物件導向程式設計的人們來說，眾多難以跨越的難關之一便是，從開始到結束太過於簡單。對於一顆久經訓練、善於找尋複雜解答的頭腦來說，往往會在接觸的一開始被這種單純特性給難倒。

當你繼承既有的 type 時，便創造了新的 type，後者不僅包含前者的所有成員（但 **priavte** 成員會被隱藏起來，而且無法存取），更重要的是它同時也複製了 base class 的介面。也就是說，所有可以發送給 base class 物件的訊息，也都同樣可以發送給 derived class 物件。由於我們可以透過「可發送之訊息型態」來得知物件的 type，因此前述事實告訴我們，derived class 和 base class 具有相同的 type。例如前一個例子中我們便可以說「圓形是一種幾何形狀」。透過繼承而發生的型別等價性（type equivalence），是了解物件導向程式設計真髓的重要關鍵。

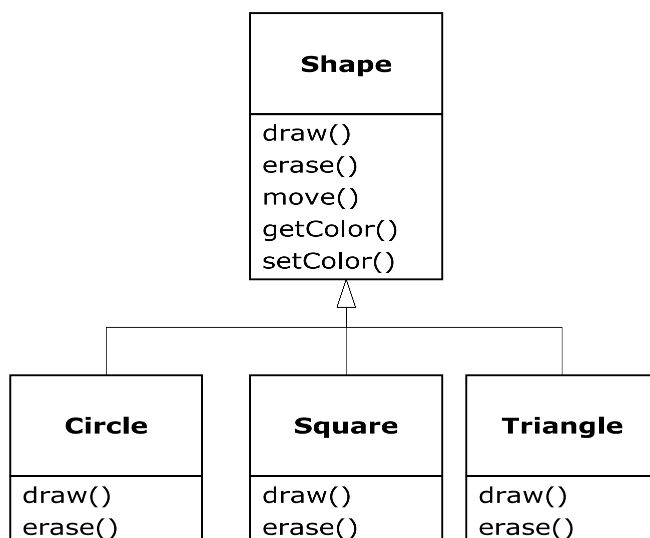
base class 和 derived class 有著相同的介面，而一定有某些實作碼伴隨著此一介面。也就是說，當物件接收到特定訊息時，還是得有程式碼來執行動作。倘若你只是很簡單地繼承了 class，然後再沒有做任何事情，那麼 base class 的介面所伴隨的函式，便會原封不動地被繼承到 derived class 去。這表示 derived class 物件不僅擁有與 base class 物件相同的型別（type），也擁有相同的行爲，這沒什麼趣味。

兩種作法可以產生 derived class 與 base class 之間的差異。第一種作法十分直覺，只要直接在 derived class 中增加新函式即可。這些新函式並非 base class 介面的一部份。這意謂 base class 無法滿足你的需要，因此你得加入更多函式。這種既簡單又基本的方式，有時候對你的問題而言是一種完美解答。但是你應該仔細思考，你的 base class 是否也可能需要這些額外功能。這種發現與更替的過程，會在整個物件導向設計過程中持續發生。



雖然繼承有時候意味著加入新功能至介面中（尤其 Java 更是以關鍵字 **extends** 代表繼承），但並非總是如此。形成差異的第二種方法（也許

是更重要的方法）便是改變既有之 base class 的函式行為，這種行為我們通常稱為「覆寫（*overriding*）」。

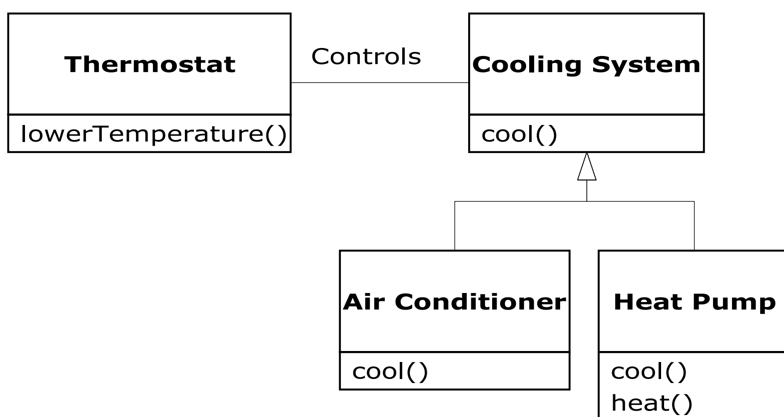


想要覆寫某個函式，只須在 derived class 中建立該函式的一份新定義即可。這個時候你的意思是：「在這裡我使用相同的介面函式，但我想在新型別中做點不一樣的事情。」

是一個（is-a）vs. 像是一個（is-like-a）

繼承過程中可以進行的動作，仍舊有些爭論。繼承應該「只」覆寫 base class 的函式（而不加入任何新函式）嗎？如果這樣，便意謂 derived class 和 base class 有著完全相同的 type，因為它們的介面一模一樣。這樣得到的結果是，你可以以 derived class 物件完全替換 base class 物件。這可視為一種「純粹替代（*pure substitution*）」，通常稱為「替代法則（*substitution principle*）」。就某種意義而言，這是處理繼承的一種理想方式。我們通常將這種介於 base class 和 derived class 之間的關係稱為「*is-a*（是一種）」關係，因為你可以說「圓形是一種幾何形狀」。套用繼承關係與否的一個檢驗標準便是，你是否可以有意義地宣稱 classes 之間具備「*is-a*」的關係。

不過有些時候，你還是得將新的介面元素加到 derived type 中，如此也就擴充了介面，進而產生新的 type。新的 type 仍然可以替換 base type，但這種形式的替換並非完美無瑕，因為 base type 無法取用你加入的新函式。這種關係我們可以用「*is-like-a*（像一個）[5]」的方式描述。新 type 具備和舊 type 相同的介面，但還包含其他函式，所以不能宣稱它們二者完全相同。以冷氣機為例，假設你的房子裝設了給所有冷卻系統用的控制機制，也就是說它具備讓你控制冷卻系統的介面。現在，冷氣機壞了，你新裝上一部冷暖氣機。這個冷暖氣機便「*is-like-a*（像是一個）」冷氣機，但它可做的事情更多。但因為房子的控制系統只能控制冷卻功能，所以只能夠和新物件中的冷卻部份溝通。新物件的介面雖然擴充了，但舊系統除了原介面之外，完全不知道任何其他事情。



當然，看過這樣的設計之後，你便會發現，base class 的「冷卻系統」不夠一般化，應該改為「溫度控制系統」，使它得以涵蓋加熱功能 — 於是我們便可套用所謂「替代法則」了。上圖是個範例，說明在設計領域和真實世界中可能發生的事情。

[5]這是我發明的詞彙。

當你了解替代法則（substitution principle），很容易便會以爲「純粹替代」是唯一可行之道。事實上如果你的設計能夠依循此種方式，是滿好的。不過偶而你還是會遭遇到「需要將新函式加入 derived class 介面」的情況。只要仔細檢閱，這兩種情形的使用時機應該是相當明顯的。

隨多型而生的可互換物件

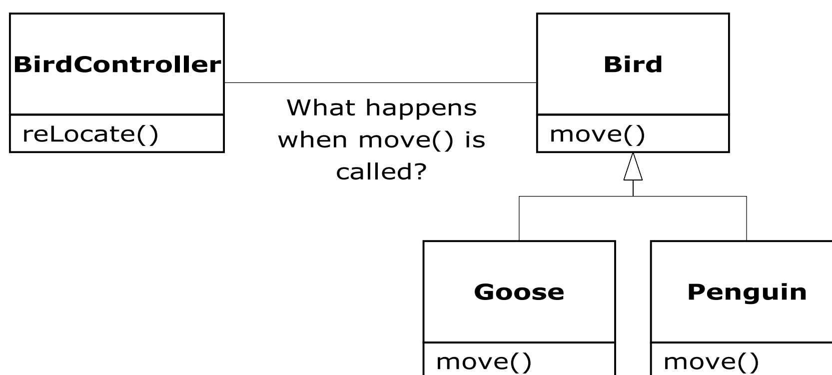
Interchangeable objects with polymorphism

處理 type 階層體系內的物件時，我們往往希望能夠不以它們所屬的特定 type 看待之，而以其 base type 視之。如此一來我們所撰寫的程式碼便不會和特定的 type 有依存關係。以幾何形狀爲例，用來操作一般化（泛化、generic）形狀的函式，其實不需要在意其所處理的形狀究竟是圓形、正方形、三角形、或其他尚未被定義的種種形狀。因爲所有形狀都可以被繪製、被擦拭、被移動，因此這些函式只需發送訊息給「形狀」物件，不需擔心對方怎麼處理這些訊息。

爲了擴充物件導向程式的能力，以便處理新狀況，最常用的手法就是加入新 types。此類程式碼的特性就是，不會因爲額外加入新型別而受到影響。例如你可以衍生出幾何形狀的 subtype – 五邊形（pentagon），卻不需要修改任何函式 — 只要這些函式僅只處理泛化的（generic）幾何形狀。這種「透過衍生新的 subtype 而擴充程式能力」的手法相當重要，因爲這種能力可以大幅改善設計，使軟體的維護成本降低。

不過，完美的事物並不存在於人間。當我們試著以泛化的 base type 來看待 derived type 物件時（例如以幾何形狀來看待圓形、以交通工具來對待腳踏車、把鸕鶿看做是鳥等等），倘若某個函式要求某一泛化形狀繪製自己，或是要求某個泛化交通工具前進，或是要求某隻泛化的鳥移動，編譯器在編譯期便無法精確知道究竟應該執行哪一段程式碼。這是關鍵所在：訊息被發送時，程式設計者並不想知道哪一段程式碼會被執行；繪圖函式施行於圓形、正方形、三角形身上完全沒有兩樣，物件執行時會依據自身的實際型別來決定究竟該執行哪一段程式碼。如果「知道哪一段程式碼將被執行」對你而言並非必要，那麼當你加入新的子型別（subtype）時，不需更動函式叫用句，就可以視子型別的不同而執行不同的程式碼。也因此，編

譯器無法精確知道究竟哪一程式碼會被執行起來。那麼編譯器又做些什麼事呢？以下圖爲例，**BirdController** 物件僅處理泛化的 **Bird** 物件，因此它「對那些 **Bird** 物件實際上是什麼型別」毫不知情。從 **BirdController** 的角度來看，這麼做是十分方便的，它將因此而不必撰寫特別的程式碼來判斷所處理的 **Bird** 物件究竟是什麼型別，也不需要判斷這些 **Bird** 物件會有什麼特別行爲。然而，在忽略 **Bird** 實際型別的情況下，當 **move()** 被呼叫時，物件的實際行爲會是什麼？鵝（**Goose**）會用跑的還是飛的？還是游泳？企鵝（**Penguin**）會用跑的還是游水的方式？



這個問題的答案，是物件導向程式設計中最重要的訣竅所在：編譯器無法以傳統方式來進行函式的叫用。由 non-OOP 編譯器所產生的函式叫用，會以所謂「前期繫結（*early binding*）」方式來呼叫函式。這個名詞你過去可能從未聽聞，因爲你從未想過能夠以其他方式來辦理。運用這種方式，編譯器對叫用動作產生出特定的函式名稱，而聯結器（linker）再將此叫用動作決議（resolves）爲「欲執行之程式碼的絕對位址」。但是在 OOP 中，程式未到執行期是無法決定程式碼的位址的，因此當我們將訊息發送給泛化物件（generic object）時，必須採用其他解決方案。

爲了解決上述問題，物件導向程式語言採用所謂的「後期繫結（*late binding*）」觀念。當你發送訊息給物件，「應被喚起的程式碼」會一直到執行期才決定下來。編譯器還是有責任確定函式的存在，並對引數（arguments）、回傳值（return value）進行型別檢驗（無法對此提供保證者，即所謂「弱型別（*weakly typed*）」語言），但編譯器仍舊無法得知究竟會執行哪一段程式碼。

爲了達到後期繫結，Java 使用一小段特殊程式碼來代替呼叫動作的絕對形式。這一小段程式碼會透過物件內儲存的資訊來計算函式實體位址（此一過程將於第七章詳述）。因此每個物件可因爲這一小段程式碼的內容不同，而有不同的行爲。當你發送訊息至某個物件，該物件便知道如何反應。

在某些程式語言裡頭（例如 C++），你得明確指出是否希望某個函式具備後期繫結的彈性。這一類語言把所有 member functions 的繫結動作預設爲「非動態」。這會引起諸多問題，所以 Java 將所有 member functions 預設爲動態繫結（後期繫結），你不需要加上任何關鍵字，就可以獲得多型（polymorphism）的威力。

回頭想想「形狀」的例子。整個 classes 族系（擁有一致介面的所有 classes）在本章稍早已有圖示。爲了說明多型（polymorphism）特性，我要撰寫一段程式碼，並在程式碼中忽略型別（types）細節，僅和 base class 溝通。這樣的程式碼和「型別特定資訊」之間已經解除耦合（decoupled）了，撰寫起來格外簡單又容易理解。舉個例子，當新型別「六邊形（**Hexagon**）」透過繼承機制加入 classes 族系時，處理舊型別的程式碼不需任何改變便可以處理新型別。也因此，我們說這個程式是可擴充的（*extensible*）。

如果以 Java 來撰寫函式（很快你就會學到如何撰寫）：

```
void doStuff(Shape s) {
  s.erase();
  // ...
  s.draw();
}
```

上述函式可以和任何 **Shape** 交談，獨立於它所繪製或擦拭的任何特定物件型別。如果我們在程式的其他地點用到了 **doStuff()** 函式：

```
Circle c = new Circle();
Triangle t = new Triangle();
Line l = new Line();
doStuff(c);
doStuff(t);
doStuff(l);
```

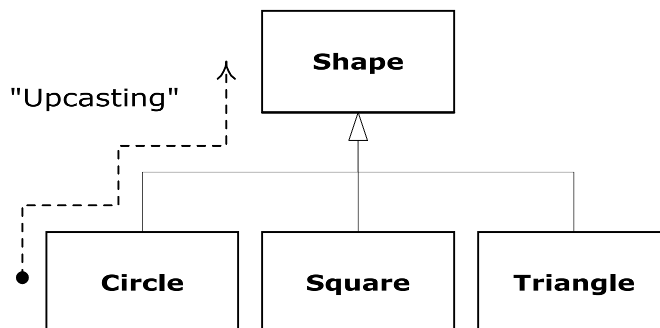
那麼當呼叫 **doStuff()** 時，不論物件之實際型別爲何，都能夠運作無誤。

這是個令人感到驚奇的手法。再看看下面這行程式：

```
doStuff(c);
```

此處當 **Circle** 被傳入這個預期接收 **Shape** 的函式時，究竟會發生什麼事呢？由於 **Circle** 是一種（*is-a*）**Shape**，所以它可被 **doStuff()** 認可。亦即「**doStuff()** 可發送給 **Shape**」的所有訊息，**Circle** 都可以接受，所以這麼做是完全安全且合邏輯的。

我們把「將 derived class 視爲其 base class」的過程，稱爲「向上轉型（*upcasting*）」。*cast* 這個字的靈感來自於模型鑄造時的塑模動作，*up* 這個字則是因爲繼承階層圖通常將 base class 置於上端而將 derived class 安排於下端，因此，轉型爲一個 base type，便是在繼承圖中向上移動，所以說是 *upcasting*。



物件導向程式一定會在程式某處做些 *upcasting* 動作，這正是你如何將自己從「執著於實際型別」救贖出來的關鍵。請看 **doStuff()** 內的函式碼：

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

注意，這裡並非表現出「如果你是 **Circle**，請做這些事；如果你是 **Square**，請做那些事…」。如果撰寫那樣的碼，你得逐一檢查 **Shape** 物

件的實際型別，那會帶來極大麻煩，因爲每當加入新的 **Shape** type，你就得改變這段程式碼。這裡所表現的意思是「如果你是 **Shape**，我知道你自己可以 **erase()**，可以 **draw()**，請妥善進行，並小心細節處不要出錯。」

doStuff() 的內容實在令人感到神奇，不知怎麼地，事情竟然能夠自己搞定。呼叫 **Circle draw()** 所執行的程式碼，和呼叫 **Square** 或 **Line** 的 **draw()** 所執行的程式碼完全不同。當 **draw()** 這個訊息發往不知所以的 **Shape** 時，竟會依據該 **Shape** 的實際型別產生正確的行爲。這真是神奇，因爲就如先前所說，當 Java 編譯器編譯 **doStuff()** 函式碼時，無法精確知道 **doStuff()** 所處理的型別究竟爲何。所以你大概會預期，它呼叫的是 base class 的 **erase()** 和 **draw()**，而不是 **Circle**、**Square** 或 **Line** 的版本。多型（*polymorphism*）是整個神奇事件的幕後推手。編譯器及執行期系統會處理相關細節，你只需知道這件事情會發生，並知道如何透過它來進行設計，這就夠了。當你發送訊息給物件時，即便動用了向上轉型（upcasting），物件仍然會執行正確動作。

抽象基礎類別與介面

Abstract base classes and interfaces

通常在一個設計案中，你會希望 base class 僅僅代表其 derived class 的介面。也就是說，你不會希望任何人產生 base class 的實際物件，而只希望他們向上轉型至 base class — 這使得其介面可以派上用場。如果這確實是你的願望，可以使用關鍵字 **abstract**（抽象的）來標示某種 class 是「抽象的」。如果有人試著爲抽象類別產生物件，編譯器會加以阻止。這是強迫某種特殊設計的好工具。

你也可以使用關鍵字 **abstract** 來描述「目前尚未實作完全」的函式，形成一種戳記（stub），表示「衍生自此 class 的所有型別，都有此一介面函式，但它目前尚無實作內容」。抽象函式僅能存在於抽象 class 之中。當 class 被繼承，抽象函式必須被實作出來，否則新的 class 仍然是個抽象 class。「建立抽象函式」讓你得以將它置於介面之中，無需被迫爲該函式提供或許毫無意義的程式內容。

關鍵字 **interface**（介面）又更進一步地發揮抽象概念，阻止任何一個函式被定義出來。**interface** 是個常被用到而又十分方便的工具，因爲它提供了「介面和實作分離」的完美境界。此外，如果你想要，你可以將多個介面組合在一起，不過你無法同時繼承多個一般的或抽象的 classes。

物件的形貌與壽命

Object landscapes and lifetimes

技術上來說，OOP 只談論抽象資料型別、繼承、多型等議題。然而其他議題也可能同樣重要，本節剩餘篇幅涵蓋了這些議題。

一個極爲重要的議題便是物件的生成（created）和毀滅（destroyed）。物件的資料存於何處？它們的壽命如何控制？這裡有一些不同的處理哲學。C++ 認爲效率是最重要的議題，因此將抉擇留給程式設計者。想取得最好的執行速度？沒問題，請將物件置於 stack（這樣的物件又稱爲 *automatic* 變量或 *scoped* 變量）或靜態儲存區中，於是程式撰寫時便決定了物件的儲存空間和壽命。這種安排是把重點擺在儲存空間的配置與釋放的速度上。某些情況下這樣的安排可能很有價值。不過這麼做卻也犧牲了彈性，因爲你必須在程式撰寫期間明確知道物件的數量、壽命、型別。如果你企圖解決的是比較一般化的問題（例如電腦輔助設計、倉儲管理、飛航管制系統等），這種方式就過於受限了。

第二種方法是從一塊名爲 heap（堆積）的記憶體中動態產生物件。在這種方式之下，除非等到執行期，否則你無法回答需要多少物件、壽命爲何、確切型別爲何…等問題。這些問題只有程式執行時方能給出答案。如果你要求新的物件，只需在必要時候從 heap 產生出來即可。由於執行期對儲存

空間的管理方式是動態的，所以從 heap 配置空間所耗用的時間，遠大於從 stack 產生儲存空間所需的時間。是的，自 stack 產生儲存空間，其動作通常只是一個簡單的組合語言指令，將 stack 指標往下移動；釋放時只要將它往上移回即可。動態法有一個前提：適用於「物件比較複雜」的情況，因此不論儲存空間的找尋或釋放時必要的額外負擔，都不致於對物件的產生造成重大衝擊。動態法帶來的彈性，正是解決一般化程式問題不可或缺的根本。

Java 全然採用第二種方法[6]。每當你想要產生物件，都得使用關鍵字 **new** 來動態產生物件的實體（instance）。

還有一個議題，就是物件的壽命。在那些「允許物件誕生於 stack 內」的程式語言中，編譯器會判斷物件應該存活多久，並可自動消滅之。但如果在 heap 之中生成物件，編譯器對其壽命將一無所知。以 C++ 爲例，你得自行撰寫程式碼來摧毀物件。因此如果不能正確做好此事，就會引發記憶體洩漏（memory leaks），這幾乎是 C++ 程式的共通問題。Java 提供了所謂「垃圾回收器（garbage collectioncollector）」機制，當物件不再被使用，會被自動察覺並消滅。垃圾回收器十分便利，因爲它可以減少你需要考量的因素，也減少你必須撰寫的程式碼數量。更重要的是，垃圾回收器提供了更高階的保障，避免隱晦的記憶體洩漏問題發生（這是許多 C++ 專案的痛處）。

本節的其餘篇幅，我們來看看諸多和物件壽命及其景貌相關的其他要素。

---

[6] 稍後你會看到，基本型別（primitive types）是特例。

群集器和迭代器

Collections and iterators

如果你不知道你將動用多少個物件，或者不知道這些物件該存活多久，才能夠解決某個問題，那麼你同樣無法知道該如何儲存這些物件。如何才能夠知道該為這些物件準備多少空間呢？這其實永遠得不到答案，因為這些資訊只有在執行期才能獲得。

相對於大多數物件導向設計問題而言，這個問題的解法似乎太過簡單：產生另一種型別的物件。這個新型物件儲存著一些 reference，代表其他物件。當然你可以使用 array 完成相同的事情，大多數程式語言都提供 array。不過這裡談的更多。這種新型物件通常被稱為容器（*container*），有時也叫群集（*collection*）。由於 Java 標準程式庫以不同的意義使用後一個術語，本書決定採用「容器」一詞。這個新式物件會適當擴展自己的容量，以便容納你置入的所有東西。你不需要知道將來會置入多少個物件於容器中，是的，只要產生容器，它會自行料理細節。

幸運的是，好的 OOP 語言往往都會伴隨一組容器，做為套件的一部份。在 C++ 中，容器是 C++ 標準程式庫的一部份，有時候也稱為 Standard Template Library（STL, 標準模板程式庫）。Object Pascal 在其視覺化元件庫（Visual Component Library, VCL）中亦提供了諸多容器。Smalltalk 提供的容器更完備。Java 也在其標準程式庫中提供了許多容器。在某些 程式庫中，通用型容器能夠完善符合所有需求，其他程式庫（例如 Java）則提供了諸般不同的容器，符合各類需求。例如 vector（向量，Java 稱之為 **ArrayList**）為所有元素提供了一致性的存取方式，linked list（串列）則提供任何位置上的元素安插動作。你應該挑選符合需求的容器。容器還可能包括：sets（集合）、queues（佇列）、hash tables（雜湊表）、trees（樹狀結構）、stacks（堆疊）等等。

所有容器都以相同的方式處理元素的置入和取出。通常它們都會提供元素安插函式，以及元素取回函式。不過元素的取出動作較為複雜，因為「只能進行單一選取動作」的函式實在是束縛過多，綁手綁腳。如果你想同時操作或比較「一組」（而非一個）元素時，怎麼辦呢？

迭代器（iterator）爲此提供了解決之道。做爲一個物件，迭代器的作用就是用來選擇容器中的元素，並將這些元素呈現給迭代器使用者。身爲一個 class，迭代器提供了某種抽象層級，可將「容器實作細節」和「對容器進行存取動作之程式碼」分離開來。經由迭代器，容器被抽象化爲僅僅是一組序列（sequence）。迭代器讓你得以走訪整個序列，無需煩惱底層結構究竟是 **ArrayList** 或 **LinkedList** 或 **Stack** 或其他。這種彈性使我們可以輕易更動底層結構而不至於干擾應用程式碼。Java 在版本 1.0 和 1.1 時代有個名爲 **Enumeration** 的標準迭代器，用於所有容器類別之上。Java 2 加入更完備的容器類別庫，其中包含名爲 **Iterator** 的迭代器，它能用於老式 **Enumeration** 力所未逮的許多事情上面。

從設計觀點來看，你需要的其實只是一個可以被操作、用來解決問題的序列。如果單單某個序列型別就可以滿足你的全部需求，實在沒有理由動用其他類型的序列。不過，基於兩個理由，你還是得面臨容器的選擇。第一，不同的容器提供了不同形式的介面和外在行爲。stack 具備的介面與行爲皆異於 queue，也和 set、list 不同。這些容器之間往往某一種會比另一種更能爲你的問題提供彈性解答。第二，不同的容器在某些操作上有著不同的效率，最佳例子便是 **ArrayList** 和 **LinkedList**。這兩種序列具備完全相同的介面與外在行爲，但在某些操作上所耗費的代價，卻有截然不同的表現。對 **ArrayList** 進行隨機存取，可以在常數時間（constant-time）完成；不論你所選擇的元素爲何，所需時間都相同。但是對 **LinkedList** 而言，「隨機選取某一元素」的動作需得在串列上行進，愈靠近串列尾端，花費的時間愈久。另一方面，如果你將元素安插至序列中央位置，對 **LinkedList** 來說花費的代價明顯少於 **ArrayList**。不同動作的效率高下，完全取決於序列的底層結構。在設計階段，也許一開始你採用 **LinkedList**，然後爲了調校系統效能，轉而採用 **ArrayList**。迭代器所帶來的抽象化，使我們在進行諸如此類的轉換時，得以將衝擊降至最低。

最後，請你牢記，容器只是個可以將物件放入的儲藏櫃。如果這個儲藏櫃可以解決你的全部需求，那麼它的實作方式就無關緊要（對大多數類型的物件而言，這是基本觀念）。如果你所處的開發環境，有著其他因素所引起的額外負擔，那麼，介於 **ArrayList** 和 **LinkedList** 之間的差別可能無關痛癢。你也許只需要一種序列型別。你甚至可以想像有一個完美的容器抽象化機制，可以根據被運用的方式，動態改變其底層實作方式。

單根繼承體系

The singly rooted hierarchy

OOP 中有個議題，在 C++ 面世之後變得格外受人注目：所有 classes 最終是否都繼承自單一的 base class 呢？Java（以及大多數 OOP 程式語言）的答案是 yes，而且這個終極的 base class 名爲 **Object**。事實證明，單根繼承體系可以帶來許多好處。

單根繼承體系中的所有物件都有共通介面，所以最終它們都屬於相同的 type。在另一種（C++ 所提供的）設計中，你無法確保所有物件都隸屬同一個基本型別。從回溯相容的觀點來看，這麼做比較符合 C 模型，並且比較不受限制。但是當你想要進行完全的物件導向程式設計時，就得自行打造自己的繼承體系，以便提供某種便利性，而這種便利性在其他語言上卻是內建的。你的程式庫內部可能會有一些不相容介面，你得額外花費力氣將新介面融入你的設計之中（天啊，面對的可能是多重繼承）。C++ 是否值得爲了前述的額外「彈性」這麼做呢？如果你需要的話 – 也許你在 C 語言上面投資頗多 – 這麼做就有價值。但如果你剛自起跑線出發，Java 這種方式通常會帶給你更大的生產力。

單根繼承體系（一如 Java 所提供）保證所有物件都擁有某些功能。在整個系統裡，你因此知道可以在每個物件身上執行某些基本操作。單根繼承體系再加上「在 heap 之中產生所有物件」，大大簡化了引數傳遞動作（這也是 C++裡頭十分複雜的問題之一）。

單根繼承體系也使垃圾回收器（內建於 Java）的實現更加容易。所有必備功能都可安置於 base class 身上，然後垃圾回收器便可發送適當訊息給系統中的每個物件。如果缺乏「單根繼承體系」及「完全透過 reference 來操作物件」的系統特性，垃圾回收器的實作就會十分困難。

由於所有物件都保證會有執行期型別資訊（run-time type information，RTTI），所以你必不會因爲無法判斷物件的確切型別而陷入動彈不得的僵局。對於異常處理（exception handling）之類的系統級操作行爲而言，這一點格外重要，並且也能爲程式設計帶來更佳彈性。

群集類別庫，及其易用性支援

Collection libraries and support for easy collection use

容器是一種常常被使用的工具，所以將容器程式庫製成內建形式，是很有意義的事情。這麼一來你就可以直接使用現成的容器，將它套入你的應用程式中。Java 提供了這樣的程式庫，而且幾乎可以滿足所有需求。

向下轉型 vs. 模板/泛型

Downcasting vs. templates/generics

爲了重複運用容器，我們會以 Java 中的某個通用型別（也就是 **Object**）爲對象，進行儲存工作。單根繼承體系意謂「萬物皆爲 **Object**」，因此，如果容器可以裝納 **Object**s，也就可以儲存任何物件。這使得容器很容易被重複運用。

欲使用這樣的容器，只需將 object reference 加入即可，稍後可以將它們取回。由於容器僅容納 **Object**s，所以當你將物件的 reference 加入容器時，會向上轉型爲 **Object**，因而遺失自己的精確身份。當你將它取回，你所得到的僅僅是個 **Object** reference，而非你所置入之物件的確切型別。現在，該透過怎樣的方式將它變回先前那個具備實用介面的物件呢？

在這裡，轉型動作再度派上用場。但這一次並非是在繼承體系中向上轉型爲更通用的型別，而是向下轉型爲更特定的型別。這種轉型動作稱爲向下轉型（*downcasting*）。你已經知道，向上轉型（例如 **Circle** 轉爲

Shape）十分安全。但由於你無法得知某個 **Object** 是否為 **Circle** 或 **Shape**，所以除非你確切知道你所處理的是什麼，否則向下轉型幾乎是不安全的。

不過向下轉型也非全然危險，因為如果向下轉型至錯誤的型別，你會得到所謂「異常（*exception*）」的執行期錯誤，稍後我會簡短介紹什麼是異常。但即便有此機制，從容器中取出 object references 時，你還是需要某種方式來記住這些物件究竟是什麼型別，才能夠放心進行向下轉型。

向下轉型及執行期檢驗動作，都會耗費額外的程式執行時間，以及程式設計者的心力。何不建立起「知道自己所儲存之物件隸屬何種型別」的容器，藉以降低向下轉型的需求及可能導致的錯誤呢？解決之道就是所謂的「參數化型別（*parameterized types*）」，這是一種由編譯器自動為我們量身訂製的 classes，作用於特定型別之上。舉個例子，如果使用這類參數化容器，編譯器便可訂製此一容器，使它只能接受（及取出）**Shape**s。

參數化型別（parameterized types）是 C++ 極為重要的組成，部份原因是 C++ 缺乏單根繼承體系。在 C++ 中，實現參數化型別的關鍵字是 **template**。Java 目前並沒有參數化型別，因為單根繼承體系已經可以涵蓋「參數化型別」的功能。不過目前有一份相關提案，其中採用和 C++ template 極為相似的語法（譯註：JSR14，目前已實現於 JDK1.4 和 GJ 身上，見 *www.jjhou.com/javatwo-2002.htm*）。

管家婆的難題：誰該負責清理？

The housekeeping dilemma: who should clean up?

每個物件為了生存，都需要動用資源，尤其是記憶體。當物件不再需要記憶體，就該加以清除，使資源可被釋放、可再被使用。在相對簡單的程式設計環境中，清理物件似乎不是什麼困難的問題：首先產生物件，然後使用之，最後它就應該被摧毀。這不難，但你可能遭遇更為複雜的情況。

舉個例子，假設你正在設計機場航空交通管制系統（同樣的模式在倉儲貨物管理、錄影帶出租系統、寵物寄宿店一樣適用）。一開始問題似乎很簡

單：做出一個容器，用來置放飛機，然後產生新飛機，然後把每一架位於飛航管制區的飛機放到容器之中。一旦飛機飛離該區域，刪去對應的物件，即是完成了清理動作。

但你或許還有其他系統，記錄著飛機的相關資訊；或許這些資料目前不會馬上爲主控功能所用。它可能記錄所有即將離開機場之小型飛機的飛行計畫。所以你會有第二個容器，用來儲存所有小型飛機；每當你產生一個小飛機物件，同時也把它置於第二容器中。然後，在程式閒置時間（idle time）以一個背景行程（background process）操作第二容器內的物件。

現在問題更加困難了：何時才是消滅這些物件的適當時機？當你用完某個物件之後，系統中的某部份還有可能再使用它。此類問題會發生在其他許多情境之中，在那些「你必須明確刪除物件」的系統中，這件事情顯得格外複雜。

Java 的垃圾回收器被設計用來處理「釋放記憶體時可能會有的種種問題」（其中並未包含物件清理動作的其他面向問題）。垃圾回收器會「知道」物件何時不再被使用，然後自動釋放該物件所用的記憶體。這種方式合併了「單根繼承體系」和「完全從 heap 配置記憶體」兩大特性，使得透過 Java 進行程式設計的過程遠較透過 C++ 單純。只需做極少量決策，只有極少的障礙需要克服。

垃圾回收器vs.效率與彈性
Garbage collectors vs. efficiency and flexibility

如果這麼做沒有任何瑕疵，爲什麼 C++ 不率先採行？是的，當然，你得爲程式設計上的便利付出代價，這個代價就是執行期的額外負擔。一如先前所述，在 C++ 中，你可以選擇在 stack 上生成物件，它們會被自動清除（但不具備彈性，無法在執行期依需要而動態產生）。在 stack 上生成

物件，對儲存空間的配置和釋放來說幾乎是最有效率的方式。在 heap 上生成物件，代價可就高昂多了。即使完全繼承同一個 base class，完全以多型（polymorphic）方式來處理函式呼叫，還是需要付出少量代價。垃圾回收器之所以特別形成癥結，問題在於你永遠不知道垃圾回收動作何時開始、持續多久。這意謂 Java 程式的執行速率會有不一致的現象，因此無法應用於某些情況下，例如在極重視程式執行速率的場合。此類程式通常稱爲即時（real time）程式 — 雖然並非所有的即時程式設計問題都如此嚴峻。

C++ 的開創者努力爭取 C 程式員的支持，而且成果豐碩，因此不希望加入任何影響速度的功能，也不期望在那些「程式員可能會選擇 C 的場合」因爲 C++ 所引進的新功能轉而採用 C++。這個目標達到了，但也使得以 C++ 進行設計時需付出「高複雜度」的代價。相對於 C++，Java 單純許多，但這種單純換來的便是效率上的折損，有時也會失去適用性。然而，對於解決大多數程式問題而言，Java 是較好的選擇。

異常處理：面對錯誤的發生

Exception handling: dealing with errors

自從天地間有了第一個程式語言，錯誤的處理始終都是最困難的問題之一。因爲良好的錯誤處理系統很難設計，所以許多程式語言乾脆直接略去這個議題，將它留給程式庫設計者，由後者提供「可用於許多情況，但是並非完全徹底」的方法，或甚至完全忽略這個議題。大多數錯誤處理系統的問題在於，它們十分依賴程式員自身的警覺性，而非語言本身的法制性。如果程式員本身不夠警覺 – 通常是因爲專案太趕 – 這些系統便容易被忽略。

「異常處理機制」將錯誤處理問題直接內嵌於程式語言中，有時甚至直接內嵌於作業系統中。所謂異常（*exception*）是一種物件，可在錯誤發生點

被擲出（throw），並在適當的處理程序中被捕捉（catch），藉以處理特定類型的錯誤。當錯誤發生，異常處理機制會採取一條截然不同的執行路徑。也正因爲如此，所以它不會干擾程式碼的正常執行。這使得程式碼的撰寫更加單純，因爲你不需要被迫定時檢查錯誤。此外，被擲出的異常、函式回傳的錯誤值、函式爲表示錯誤發生而設定的旗標值，三者之間有著本質上的差異 － 後二者都可以被有意忽略，異常則無法被忽略，保證一定會在某處被處理。最後一點：異常爲「錯誤情境的確實回復」提供了一種可行方案，是的，不再只能選擇退出（那是一種逃避），如今你可以校正事情，回復程式的執行，這種表現是穩健而強固的程式的特質。

Java 的異常處理機制在眾多程式語言中格外引人注目，因爲 Java 一開始就將異常處理機制內嵌進來，強迫你使用。如果你沒有撰寫可適當處理異常的程式碼，便會發生編譯錯誤。這種一致的態度使得錯誤處理更加容易。

雖然物件導向程式語言常以一個物件表現一個異常（exception），但異常處理機制並非物件導向的特性。是的，異常處理機制在物件導向語言出現之前，存在久矣。

多執行緒

Multithreading

電腦程式設計有一個很基礎的觀念，那就是必須能夠同時處理多個工作（task）。許多設計上的問題，需得程式停下手頭的工作，處理一些其他事情，再返回主行程（main process）。辦法很多，早期程式員透過對機器的低階認知，撰寫中斷服務常式（interrupt service routines），透過硬體中斷的觸發，暫停主行程的執行。雖然這麼做沒有問題，但這種作法難度較高，也不具可攜性。

有時候，在處理「時間因素極爲關鍵」的工作（task）時，中斷的使用是必要的。但還有其他爲數不少的問題，只需將問題切割爲多個可獨立執行的片段，便能夠讓整個程式更具反應力。這些「可獨立執行的片段」便是所謂的執行緒（threads），而上述這種觀念便被稱爲「多緒」（multithreading）。最常見的例子便是使用者介面的運作了，透過執行緒的使用，雖然還有某些處理動作正在電腦中進行，使用者仍然可以按下按鈕，不受阻礙。

通常，執行緒只是一種用來「配置單一處理器執行時間」的機制。但如果作業系統支援多處理器的話，不同的執行緒便可以被指派至不同的處理器執行，真正做到並行（parallel）執行。在程式語言這個層次上提供多執行緒，所能達到的便利之一便是，讓程式設計者無需考量實際機器上究竟存在多少個處理器。程式只是邏輯上被劃分爲多個執行緒，如果機器擁有多個處理器，不需特別的調整，程式就能夠執行得快一些。

這使得執行緒聽起頗爲單純，但當發生資源共享時，卻會出現隱約的問題。倘若多個並行的執行緒共用同一份資源，就會引發問題。舉例來說，兩個行程（processes）無法同時送出資訊至單一印表機。爲了解決這個問題，某些可被共享的資源（例如印表機），必須在使用時加以鎖定。因此整個過程是：執行緒鎖住某資源、完成自己的工作、解除鎖定讓其他行程有權力使用該資源。

Java 在程式語言中內建了執行緒功能，讓此一複雜課題變得單純。由於執行緒功能是在物件層次上提供，因此一個執行緒便以一個物件來表示。Java 也提供有限的資源鎖定功能，可以鎖定任何物件所用的記憶體（也算是某種形式的資源鎖定），使得同一時間內只有一個執行緒可使用這個物件。透過關鍵字 **synchronized** 便可達到此一目的。其他型態的資源就必須靠程式員自行鎖定，通常的作法是產生一個物件，代表欲鎖定的資源；所有執行緒在存取這份資源之前都必須先加以檢驗。

永續性

Persistence

物件生成之後，只要你還需要它，它就會持續存在。但是一旦程式結束，它就不再有生存環境了。如果物件可以在程式非執行狀態下依舊存在、依舊保有其資訊，那麼在許多應用中將大有幫助。因爲當程式重新啓動，物件便又能夠復活，而且仍然具備上次執行時的狀態。當然，你可以簡單地將資料寫到檔案或資料庫，進而達到這個效果。但是在「萬事萬物皆物件」的精神下，如果能夠將物件宣告爲永續的（persistent），而且由語言系統自動爲你處理所有相關細節，不就太好了嗎？

Java 提供所謂的「輕量級永續性（light weight persistence）」，讓你可以很輕鬆地將物件儲存於磁碟，並於日後取回。稱「輕量級」的原因是，你還是得自己呼叫儲存、取回動作。除此之外，JavaSpaces（第十五章敘述）還提供另一種永續儲存功能。未來版本可能還會提供更完整的支援。

Java 與 Internet（網際網路、互聯網）

如果 Java 不過是程式語言芸芸眾生中的一個，你可能會問，爲什麼它如此重要？爲什麼會被宣傳爲電腦程式設計革命性的一大步。從傳統程式設計的觀點來看，答案並不那麼明顯。雖然 Java 在解決傳統的個別程式設計問題上非常有用，但更重要的是，它能夠解決 World Wide Web（全球資訊網，萬維網）上的程式設計問題。

Web 是什麼？

Web 一詞，就像其他諸如 surfing（網絡衝浪）、presence、home pages（首頁）等詞彙一樣，乍見之下可能難以理解。對於 Internet 這波狂潮，愈來愈強的反向力量在質問著，這波其勢難擋的變動，經濟價值和結果究竟爲何。如果我們回頭審視其真實面貌，應該會大有幫助。要這麼做，就得先從所謂的「主從（client/server）」系統開始了解。這個系統正是電算技術中另一個令人充滿諸多困惑的東西。

主從式計算技術
Client/Server computing

主從式系統的核心概念是，系統中存在一個集中的資訊貯存所，貯存某種形式的資料，通常位於資料庫中。你想要將這些資訊依據需求，傳佈給某些人或某些機器。主從概念的關鍵核心便在於，資訊貯存的位置集中於一點。因此每當該點的資訊改變，變動結果便會傳播至資訊使用者手上。綜合言之，資訊貯存處是一套軟體系統，能將資訊傳播出去，而資訊和軟體系統所在的機器（可能是單部機器，可能是多部機器）便稱之爲伺服器（*server*）。位於遠端機器上、和伺服器溝通以便擷取資訊、處理資訊、顯示資訊的軟體系統，便稱爲用戶端（*client*）。

主從式計算的基本觀念並不複雜。問題出在你只有單一伺服器，卻必須同時服務多個用戶。一般而言這會動用所謂的資料庫管理系統，讓設計者得以安排「資料置於表格（table）中的佈局方式」，藉以取得最佳使用效果。除此之外，系統通常也允許用戶增加新的資訊至伺服器。這意謂你必須確保某個用戶的新資料不會覆蓋另一個用戶的資料，並確保資料在加入資料庫的過程中不會遺失，此即所謂的交易處理機制（transaction processing）。一旦用戶端軟體有所改變，它們必須被設置、被除錯、然後被安裝於用戶端機器上。整個過程可能超乎你想像的複雜與費力。如果想同時支援多種不同類型的電腦或作業系統，事情更是格外麻煩。最後還有一個重要課題：效率。是的，可能有數以百計的用戶同時向伺服器發出請求，因此即便是一點小小延遲，都可能帶來重大影響。爲了降低延遲，程式員必須努力分散欲處理的工作，通常是分散至用戶端機器，但有時候會使用所謂的中介軟體（*middleware*），分散至伺服端的其他機器。中介軟體也被用來改善系統的可維護性（maintainability）。

「將資訊分散至多處」的這個單純想法，實作上卻有極多層次的複雜度，整個問題簡直是棘手得令人絕望。然而主從運算幾乎主宰了半數的程式設計活動。從訂貨、信用卡交易到各種形式 – 股市、科學、政府、個人 –

的資料傳佈。過去以來的路線是，每面對一個新問題，就發明一個新的解決方法。這些方法都很難開發，很難使用，使用者必須一次又一次學習新的介面。主從架構下的問題需要一個全面性的解決。

Web 就是一個巨型伺服器
The Web as a giant server

整個 Web 體系，實際上就是個巨大的主從系統。更糟的是，所有的伺服器與用戶端同時共存於同一個網絡中。不過你沒有必要知道這件事，雖然你可能會爲了搜尋想要的伺服器而遍尋全世界，但同一時間你只要考慮單一伺服器的連接與互動就好了。

剛開始只是很單純的單向過程：向伺服器提出請求，然後伺服器回傳一份檔案，機器上所執行的瀏覽器（一種用戶端軟體）便根據本機（local machine）上的格式來解讀這份檔案。但是很快地，人們開始進行更多嘗試，不單從伺服器擷取檔案，還希望擁有完整的主從架構能力，讓用戶端也能將資訊回饋至伺服端，例如在伺服器上進行資料庫的搜尋、將新資訊加入伺服端、下訂單（所需的安全性比早先系統所能提供的還要高）等等。這樣的歷史變革，是我們可以從 Web 的發展過程中觀察到的。

Web 瀏覽器的概念更是向前跨了一大步：同一份資訊可以不需任何改變，便顯示於各種類型的電腦上。但瀏覽器仍然過於原始，很快便因爲加諸其上的種種需求而陷入困境。是的，瀏覽器缺乏完備的互動能力，不論你想做什麼事情，幾乎都得面對「將資訊送回伺服器去處理」的問題。也許花費了幾秒鐘、甚至幾分鐘之後，才發現你所輸入的資料拼字錯誤。這是因爲瀏覽器只是一個觀看資料的工具，無法執行任何即便是最簡單的計算工作。從另一個角度看，這樣倒也安全，因爲這樣就無法在你的本機上執行任何可能帶有臭蟲或病毒的程式。

數種不同的方法可以解決這個問題。首先是改善圖形標準規格，因而得以在瀏覽器中撥放較佳的動畫和視訊。其餘問題必須透過「在瀏覽器上執行

程式」的方式加以解決。此即所謂「用戶端程式開發（client-side programming）」。

用戶端程式開發

Client-side programming

Web 最初的「伺服器－瀏覽器」設計方式，可提供互動性內容，但這種互動完全由伺服器提供。伺服器產生靜態的頁面內容，瀏覽器簡單地加以解讀，然後顯示。基本的 HTML 包含有資料收集機制：文字輸入方塊（text-entry boxes）、核取方鈕（check boxes）、選取圓鈕（radio boxes）、列式清單（lists）、下拉式清單（drop-down lists）等等。此外還包括按鈕（button）：可用於表單（form）資料的清除或提交（submit，傳給伺服器）。這種提交動作是透過任何一部 Web 伺服器都會提供的 CGI（Common Gateway Interface，共通閘道介面）達成。提交內容會告訴 CGI 要做些什麼工作。最常見的動作便是執行伺服器上某個目錄底下的某個程式，該目錄通常被命名為 "cgi-bin"。（按下 Web 頁面上的按鈕後，如果你觀察瀏覽器上方的位址視窗，有時候便會看到 "cgi-bin" 字樣，後面跟著一大堆冗長不知所云的東西。）幾乎所有程式語言都可用來撰寫這種類型的程式，Perl 是最常見的選擇，因為它被設計用來處理文字，而且是解譯式語言，因此無論伺服器所使用的處理器以及所安裝的作業系統是什麼，Perl 都可以安裝在上面。

今天，許多頗具影響力的 Web 站台，完全以 CGI 打造。事實上你幾乎可以用它來達成所有目的。但是以 CGI 程式建構的 Web 站台可能很快會變得過於複雜而難以維護，並衍生「回應時間過長」的問題。CGI 程式的回應時間和傳送資料量的多寡有關，也和伺服器的負載以及網絡的擁擠程度有關。而且 CGI 程式的初始化動作先天上就比較慢。Web 初始設計者並沒有預見到，網絡頻寬（bandwidth）是如此快速地被人們所發展出來的眾多應用程式耗盡。因此任何形式的動態繪圖動作幾乎都不可能，因為得先產生一個個 GIF 檔案，再一個個從伺服端搬至用戶端。此外你一定也處理過「驗證表單資料」之類的簡單事情：你在某個頁面上按下 "submit"（提交） 鈕，資料便回送至伺服器，然後伺服器執行某個 CGI 程式；這個程式

查覺輸入資料的錯誤，產生一份 HTML 頁面通知你有誤，並將此頁面回傳給你；於是你回到前一頁面，再試一次。這樣不僅十分緩慢，也不優雅。

這個問題的救星就是用戶端程式開發（client-side programming）。大多數執行 Web 瀏覽器的機器，其實都威力十足，可堪執行龐大的工作。在原始的靜態 HTML 方式下，這些機器只是杵在那兒等著伺服器送來下一個頁面。「用戶端程式開發」所指的，便是利用 Web 瀏覽器來執行某些它能夠執行的工作，讓網站使用者覺得運行更迅速，操作介面更具互動性。

用戶端程式開發的問題在於，它和一般的程式開發極不相同。參數幾乎相同，平台卻有差異：Web 瀏覽器就像一個功能受限的作業系統。當然，最後你還是得寫程式，而且還是得處理那些令人頭暈眼花的問題，並以用戶端程式開發方式來完成解法。這一節的剩餘部份，我便概觀性地討論用戶端程式開發中的諸般問題和解法。

Plug-ins

用戶端程式開發所跨出的一大步，便是發展所謂的 "plug-in"。透過這種作法，程式員得以下載一小段程式碼來為瀏覽器加入新功能。那一小段程式碼會將自己安插到瀏覽器的適當位置。它會告訴瀏覽器：從現在開始，你可以執行這種新功能。於是某些快速、具威力的功能便可透過 plug-in 加到瀏覽器上。你只需下載 plug-in 一次就好。撰寫 plug-in 並不是件輕鬆的事，也不是為特定網站而做的事。對用戶端程式開發而言，plug-in 的價值在於它讓專家級程式員發展新形語言，並將該語言加至瀏覽器中，而不須經過瀏覽器製造商的許可。因此，plug-in 提供了所謂的後門（back door），藉以加入新的用戶端程式語言。當然啦，並非所有的用戶端程式語言皆以 plug-in 實作。

Scripting（描述式、劇本式）語言

Plug-in 帶來了 script 語言的蓬勃發展。透過 script 語言，你可以將用戶端程式的原始碼直接內嵌於 HTML 頁面中。一旦 HTML 頁面被顯示，負責解譯該語言的 plug-in 便會被自動喚起。script 語言先天上比較容易理解，而且它們只是 HTML 頁面內的部份文字，所以當伺服器收到請求（request）而欲產生 HTML 頁面內容時，可以很快加以載入。犧牲的是你的程式碼的隱密性。不過通常我們不會使用 script 語言做過於複雜的事情，所以隱密性的問題不算太嚴重。

這也點出了 script 語言的缺點：只能在 Web 瀏覽器中拿來解決特定型態的問題，更明確地說是指用來建立更豐富、更具互動性的圖形使用介面（GUIs）。script 語言的確可能解決用戶端程式開發過程中遭遇的百分之八十問題。如果你的問題恰恰落在這百分之八十之中，script 語言可爲你提供簡單快速的開發過程。在考慮使用諸如 Java 或 ActiveX 之類更複雜的解法之前，或許你應該先考慮 script 語言。

最常被討論的瀏覽器 script 語言包括：JavaScript（和 Java 毫無關連；其命名只是爲了搭上 Java 浪潮而已）、VBScript（程式風貌和 Visual Basic 相彷）、Tcl/Tk（最受歡迎的跨平台 GUI 程式設計語言）。還有一些 script 語言不在此列，有一些正在發展之中。

JavaScript 或許是其中最被廣泛支援的一種。Netscape Navigator 和微軟的 Internet Explorer（IE）都提供對它的內建支援。此外，比起其他瀏覽器語言，市面上的 JavaScript 書藉或許是最多的。有許多自動化工具可以使用 JavaScript 自動產生頁面。不過如果你已經對 Visual Basic 或 Tcl/Tk 駕輕就熟，熟練地運用它們應該會比學習另一個新語言更具生產力，因爲你可以將全部心力集中於解決 Web 相關問題。

Java

如果 script 語言可以解決用戶端程式開發問題中的百分之八十，其他的百分之二十（那些真正很難的東西）該怎麼辦？現今最受歡迎的解決方案便是 Java。不只因爲它是個威力十足的語言，內建了安全功能、跨平台能力、提供易於進行國際化動作的工具，而且更重要的是，Java 持續擴充其語言功能和其程式庫，得以優雅地處理傳統語言中諸多未能處理好的問題，像是多緒、資料庫存取、網絡程式開發、分佈式計算。Java 係透過所謂的 *applet* 來進行用戶端程式開發。

所謂 applet 是個小程式，只能執行於 Web 瀏覽器上。做爲 Web 頁面中的一部份，applet 會被自動下載（就像頁面中的圖形會被自動下載一樣）。applet 被激化（*activated*）之後便開始執行，這是它優雅的地方之一 – 當使用者需要用戶端軟體時，便能夠自動將用戶端軟體從伺服器分發出去，不需事先安裝。使用者絕對可以取得最新版本的用戶端軟體，不會安裝錯誤，也不需要困難的安裝過程。由於 Java 的這種設計，程式員只需發展單一程式，該程式便能夠自動運作於所有電腦之上 — 只要這些電腦裝有「內建 Java 解譯器」的瀏覽器即可（大多數機器都如此）。由於 Java 是個發展完備的語言，所以在將請求送往伺服器之前和之後，你都可以盡情地在用戶端工作。例如你不必先透過網絡送出請求表單（form），而後才發現所填的日期或參數錯誤。用戶端的電腦也可以快速繪出資料，不必等待伺服器完成圖形繪製後才將圖形傳回。你所獲得的不僅是高速度和高回應能力，也可以降低網絡流量和伺服器負載，不至於拖累整個 Internet 的運作速度。

Java 凌駕於 script 語言之上的另一個優點就是，它是編譯式語言，所以用戶端無法看到原始碼。雖然不需花太多力氣就可以對 Java applet 進行反譯動作，但程式碼的隱藏通常不是什麼重要課題。此外，還有兩個重要因素。一如稍後所見，編譯過的 Java applet 可以由多個模組構成，必須引發

多次對伺服器的存取動作（hits）才可以下載這些模組。（在 Java 1.1 以及更新版本中，透過 Java 保存檔 — 所謂 JAR — 使得所有必要模組都能夠以壓縮形式包裝起來，因此下載單一檔案便可取得所有模組。）script 程式只需以文字形式整合到 Web 頁面內即可（通常比較小，也減低對伺服器的存取），這對 Web 站台的回應速度很重要。另一個因素是極爲重要的「學習曲線」。不論你過去所學爲何，Java 都不是個容易學習的語言。如果你是 Visual Basic 程式員，那麼投向 VBScript 可能是你的最快解決方案，它或許可以解決大多數普通的主從式架構問題，這些問題很難拿來印證 Java 的學習。如果你過去對 script 語言已有經驗，那麼付諸 Java 之前，先看看 JavaScript 或 VBScript 會對你大有幫助，因爲它們可能更容易符合你的需求，而且使你更具生產力。

ActiveX

就某種程度而言，Microsoft ActiveX 和 Java 比起來雖然骨子裡是完全不同的技術，卻足堪與 Java 競爭。ActiveX 原先僅用於 Windows 系統上，但透過獨立的聯合性組織，其跨平台能力正在成長。ActiveX 主張，如果你的程式只連接於自身所屬的環境，那麼此程式便可直接內含於 Web 頁面內，並在支援 ActiveX 的瀏覽器上執行 — IE 直接支援 ActiveX，Netscape 則需透過 plug-in 才能辦到。因此 ActiveX 並不侷限於特定的程式語言。舉個例子，如果你曾使用 C++、Visual Basic、Borland Delphi 等語言，在 Windows 平台上進行程式開發，那麼幾乎不需改變任何程式設計知識，就可以產生 ActiveX 組件（components）。ActiveX 也讓你得以移轉舊有程式碼至 Web 頁面中。

安全性 Security

「自動透過 Internet 下載程式並執行」聽起來簡直就是病毒作者的夢土。ActiveX 特別帶起用戶端程式開發中關於安全性的議題。當你點擊某個 Web 頁面，可能會自動下載許多東西：GIF 檔案、script 程式碼、編譯過的 Java 程式碼、ActiveX 組件…。某些東西可能是良性的，例如 GIF 檔就不會造成傷害，script 語言能做的事情通常也都十分有限。Java 的設計理念中，也將 applet 的執行侷限於受到安全保護的砂盒（sandbox）內。在

砂盒裡頭，applet 無法對磁碟寫入資料，也無法存取砂盒外的記憶體內容（譯註：砂盒是西方小孩普遍的玩具，砂子只能在砂盒裡頭，不能溢出）

ActiveX 剛好位於光譜的另一端。ActiveX 程式開發就和一般的 Windows 程式設計一樣，百無禁忌。如果你點擊了某頁面，因而下載了某個 ActiveX 組件，此一組件便有可能破壞磁碟中的檔案。這是當然啦，只要你所載入的程式不受限於 Web 瀏覽器，便可以玩出相同的戲碼。來自電子佈告欄系統（BBSs）的病毒，長久以來一直都是嚴重的問題，Internet 無遠弗屆的速度，更強化了這個問題。

所謂「數位簽證（digital signatures）」似乎是解答所在。透過數位簽證，我們可以檢驗程式作者究竟是誰。這個方法著眼於病毒的運作方式 – 病毒作者永遠是匿名的，如果能夠消除匿名行爲，那麼每個人都必須爲自己的行爲負責。乍看之下不錯，因爲這麼做可以讓程式更加實用。但我懷疑這麼做是否真能消除所有的惡意禍害。而且如果某個程式有臭蟲，雖非故意造成破壞，仍然會導致問題發生。

透過砂盒（sandbox）的作法，Java 可以避免上述問題發生。存於本機 Web 瀏覽器之中的 Java 解譯器（intepreter），在載入 applet 的同時便檢驗 applet 是否含有不適宜的指令。因此 applet 無法將檔案寫入磁碟，也無法刪除檔案（而這些正是病毒的生存方式與爲害方式）。applet 通常被視爲安全，而「安全」正是可靠的主從系統的支柱。Java 語言中任何可能滋生病毒的缺點，都會被迅速修正。（瀏覽器軟體會強制施行這些安全限制，某些瀏覽器甚至允許你選擇不同的安全等級，藉以提供不同等級的系統存取能力。譯註：不僅瀏覽器可以這麼做，所有 Java 程式都可以透過自訂 SecurityManager 的安全策略而辦到。）

你可能會質疑這個方法太過嚴苛，因爲這樣就無法將檔案寫入本機磁碟。舉個例子，你可能會想建立本機資料庫，或是將檔案儲存起來，留待離線時使用。然而，即便最初的夢想是要讓每個人最終都可以在線上進行所有重要工作，但大家馬上了解到這可能不切實際（直到有朝一日，低成本的網絡設備可以滿足大部份使用者的需求）。簽證過的 applet 可以回應你的質疑。「公開金鑰加密」（public-key encryption）可以讓我們檢驗 applet

是否真的出自它所宣稱的來源。雖然簽證過的 applet 仍然可能毀掉你的磁碟，但你現在可以相信 applet 的作者擔保他們不會進行惡意動作。Java 為數位簽證提供了完整的架構，所以你還是可以在必要的時候讓 applet 踏出砂盒的牢籠。

數位簽證遺漏了一個重要問題，那就是人們在 Internet 上的行動速度。如果你下載某個有問題的程式，而且該程式執行了某些不適宜的動作，需要多久時間才能發現這個破壞？可能數天，也可能長達數週。到那時候，如何追蹤究竟是哪個程式造成的破壞呢？而它又究竟在哪個時候幹了些什麼好事？

Internet vs. intranet

主從架構的問題，目前最常被採用的解決方案就是 Web。所以，即使只想解決此類問題中的一個子集 — 更明確地說是同一公司內的主從架構上的典型問題 — 一樣可以使用 Web 技術。如果採取傳統的主從架構解法，你得處理因用戶端電腦平台不同而衍生的種種問題，也得面臨安裝新的用戶端軟體的種種難處。上述兩種問題都可以透過 Web 瀏覽器和用戶端程式開發來解決。當 Web 技術僅用於特定公司的資訊網絡時，我們將此網絡稱為 intranet（企業內部網絡）。Intranet 相較於 Internet 能夠提供更高的安全性，因為你可以實際控制對公司內部伺服器的存取動作。從教育訓練的角度來看，當人們了解瀏覽器的一般概念之後，便很容易可以處理網頁和 applet 外觀上的差異，進而降低新系統的學習曲線。

安全問題把我們帶到一個用戶端程式開發世界自動形成的角落。假若你沒有把握你的程式會被執行於 Internet 上的什麼平台，那麼你就得格外小心，別散播含有臭蟲的程式。你需要的是能夠跨平台、具安全性的程式語言，例如 script 語言或 Java。

如果執行環境是 intranet，考量條件就不同了。一個企業內的所有機器都採用 Intel/Windows 平台，是常有的事。在 intranet 中你得負責你自個兒的程式碼品質，並在發現問題之後加以修正。除此之外，你可能也有以往

用於傳統主從架構解決方案的程式碼，每次更新時都得重新安裝用戶端程式。要知道，移轉至 Web 瀏覽器的一個主要原因便是爲了減少升級版本的安裝耗費時間。因爲，透過瀏覽器，所有升級動作都被隱藏起來，並自動進行。如果你的應用範圍是在這樣子的 intranet 中，那麼最具意義的解法便是選擇一條「得以使用既有程式碼」的路線，而不是以一個新語言重新撰寫它們。

如果你覺得很難從這麼多用戶端程式開發策略中進行選擇，最好的策略便是進行成本效益分析。考量待解問題的諸般限制，再研判何種解法最直接、最省力。即便採取了用戶端程式開發的路線，你還是得設計程式。針對自己的應用情境找出最快的發展方式，永遠都是正途。爲那些「程式發展中不可避免必然會碰觸的問題」先做準備，是積極的態度。

伺服端程式開發

Server-side programming

截至目前我完全沒有對伺服端程式開發進行探討。當你對伺服器發出請求，究竟會發生什麼事？大部份時候發出的請求只是簡單如「請傳給我某個檔案」，你的瀏覽器然後便以某種適當方式來解釋這個檔案的內容：可能是 HTML 頁面、圖形影像、Java applet、script 程式等等。某些更複雜的請求可能希望伺服器進行資料庫交易。常見的情形便是請求伺服器進行複雜的資料庫搜尋動作，然後由伺服器將結果編排於 HTML 頁面中，再回傳給你。當然，如果用戶端因爲 Java 或 scrip 語言而更具聰明相，伺服端可以只傳遞原始資料，然後在用戶端進行頁面編排動作，如此可以加速處理速度，降低伺服器的負載。另一種情況是，當你加入某個團隊或下了某個訂單，你可能會想要在資料庫中註冊你的名字，而這得更動資料庫的內容。如此一來便得透過某些伺服端程式碼，處理這些對資料庫動作的請求，這便是所謂的「伺服端程式開發（server-side programming）」。過去，伺服端程式開發通常採用 Perl 或 CGI，現在則出現了更爲複雜的系統。其中有一種是透過 Java-based Web 伺服器，讓你以 Java 語言撰寫所謂的 servlets 程式。servlets 及其衍生產物 JSP，是許多公司在網站系統的

發展上轉向 Java 的主要原因，尤其是它們可以消除因瀏覽器能力不同而衍生的問題。（譯註：所謂 servlet 一詞是由 "server" 和 "let" 兩個字合成。Java 世界常以字尾 "let" 表示小東西，例如 applet 是由 "application" 和 "let" 合成，表示小型應用程式。servlet 代表小型伺服端程式）

另一個截然不同的戰場：傳統系統

儘管 Java 骨子裡是個通用性程式語言，有能力解決所有類型的問題（至少理論上如此），但 Java 引起世人的興趣大部份始於 applet。正如先前所指出，目前還存在許多有效方法，可以解決大多數主從架構下的問題。當你離開了 applet 戰場（當然也就從諸多限制中解放了出來，例如從此可以對磁碟寫資料），便進入了通用性應用系統的世界。在這個世界裡，應用程式都是獨立執行，不須仰賴 Web 瀏覽器之鼻息，和一般程式沒有什麼兩樣。Java 的威力不僅僅來自於它的可攜性，也來自於它的「可程式化能力（programmability）」。就在你閱讀此書的同時，Java 已具備許多功能，讓你在更短的時間週期內（勝過任何一種語言）開發穩固的程式。

不過，請你了解，魚與熊掌不可得兼，開發速度是以執行速度爲代價（當然也有許多努力正著眼於這一點，尤其是 JDK 1.3 引入了所謂的 hotspot 技術，能夠大幅改善執行效能）。就像其他語言一樣，Java 具備某些先天上的限制，使它不適合用來解決某些類型的問題。雖然如此，但 Java 的演化成長極爲迅速，每個新版本都讓它在解決更多類型的問題上，具備更高度的吸引力。

分析與設計

物件導向模式，對程式設計而言是一種全新而截然不同的思考方式。第一次接觸 OOP 專案時，許多人都可能感到困擾。一旦你知道每項事物都被想成是物件，而且學會了如何以物件導向的思考方式來思考之後，你便有能力利用 OOP 帶來的優勢，開始進行「良好」的設計。

方法（或方法論，*methodology*），是一組過程和啓發，用來分解程式設計問題的複雜性。物件導向程式設計誕生之初，已有許多系統化的 OOP 方

法論存在。本節便讓你感受一下，採用此類方法來解決問題，是怎樣一種滋味。

「方法論」是充滿實驗性質的一個領域，特別是在 OOP 之中。因此，採納某種方法之前，了解方法本身所欲解決的問題本質，格外重要。對 Java 來說更是如此。Java 係用來降低程式表達上的複雜度（相較於 C 而言）。這或許可以降低我們對於那些恒常複雜之方法論的需求。在 Java 中，透過簡單方法所能處理的問題模型，比起在程序式語言（procedural languages）中採用同樣簡單之方法所能處理的問題模型，要大得多。

請注意，「方法論」一詞通常太過偉大，而且承諾過多。其實設計和撰碼時所做的事情，就是方法。你可能用的是自己發明的方法，而不自知那正是所謂的方法，但它的確是你所建立、所經歷的一個過程。如果它是個有效的方法，可能只需小幅調整，就可套用於 Java 身上。不過，如果你不滿意自己的生產力、不滿意程式的執行結果，你可能會考慮採用正式的方法，或者自眾多正式方法中採擷片段來使用。

一旦你經歷了數個開發流程，最重要的事莫過於：別放棄，做起來其實很簡單。大多數分析和設計方法，都設定在大型問題的解決方案中。但是請記住，大多數專案並不落在這個範圍內，所以你可能是在一個遠比方法論所建議的規模爲小的子集中，成功地分析和設計[7]。某些類型的程序，不論受到怎樣限制，先經過分析設計往往比直接就開始寫碼高明多了。

這很容易流爲一種形式上的固執情結，掉進所謂的「分析導致的癱瘓（analysis paralysis）」中。你會因爲無法在現階段確定每個細節，而覺得無法往前移動。請記住，不論做了多少分析，還是會有些許系統上的問題，非得等到設計時期才會顯露；有更多問題非得等到真正撰寫程式碼或

---

[7] Martin Fowler 所著的《*UML Distilled 2nd edition*》（Addison-Wesley, 2000）之中有極佳例子，說明如何將一個「有時難以抗拒」的 UML process 降低爲一個可管理的子集。

甚至程式執行時，才能夠發覺。正因如此，快速完成分析和設計動作，並實作出測試系統，是相當重要的。

這一點值得特別強調，因爲程序式語言（procedural language）帶給我們的歷史因素，使我們以爲，一個團隊在進入設計和實作階段之前，必須周延地考慮任何瑣碎細節，並善加了解。無疑地，發展 DBMS（譯註：Database Management System，資料庫管理系統）時得徹底了解客戶的需求，但是像 DBMS 這樣的問題正是那種「形式已定、已被充份了解」的問題。在這樣的程式中，資料庫結構才是重點。本章所探討的問題類型，卻是我所謂「*wild-card*」之類的問題。其解法並非單靠「套用既有解法，換湯不換藥」就可獲得，而是牽扯到一個或多個所謂「wild-card」因子 – 對於這些因子，目前沒有任何已被充份了解的解法，因此有研究的必要[8]。任何人如果企圖在設計和實作階段前徹底分析「wild-card」問題，都只會落入「分析導致的癱瘓」下場，因爲設計階段缺乏足夠的資料用來解決此類問題。想要解決此類問題，必須在整個開發週期中不斷來回往返，並採取冒險行爲（這是合理的，因爲你所進行的是新事物，而其回報也較高）。企圖在短期間獲得初步實作，看起來似乎會提高風險，但它反而能夠在 wild-card 專案中降低可能的風險。因爲你會提早發現某個特定方法對問題的解決是否可行。專案的發展，其實就是風險的管控。

我們常常聽到人家說「建造一個，用完丟掉」。透過 OOP，你可能只需丟掉「部份」即可，這是因爲程式碼被封裝爲許多 classes。第一個階段，你必然會設計出某些有用的 classes，並發展出某些在系統設計上具有價值的想法，而這些 classes 以及這些想法是不需要被丟棄的。因此，在問題的解決上，快速的第一階段所做的不僅僅只是生產出「對後繼的分析、設計、實作階段來說十分重要的資訊」，同時也必須建立程式碼的發展根基。

---

[8] 我用來評估此類專案的首要原則是：如果存在一個以上的 wild-card，那麼在建立足堪運作的雛形系統之前，甚至不要嘗試規畫整個專案會花費多少時間、耗用多少成本。

也就是說，如果你正在檢視某個方法論，其中涵蓋十分龐大的細項內容，並主張使用許多步驟和文件，那麼想要知道何時才能停止，仍然極其困難。請記住，你得試著找出以下事物：

1. 所用的物件是什麼？（如何將案子切割爲眾多組成？）
2. 物件的介面是什麼？（你得發送什麼樣的訊息給每個物件？）

確定了物件及其介面之後，就可以開始撰寫程式了。基於某些理由，你可能還需要更多的描述和文件，但以上兩點是最基本的需求。

整個開發過程可以劃分成五個階段，階段 0 就是對某種結構的運用做最初的確認。

階段 0：策劃

Phase 0: Make a plan

首先你得決定你的程序（process）將涵蓋那些步驟。聽起來很簡單（事實上所有動作聽起來都很簡單），但是在開始撰寫程式碼之前，人們通常不會進行這樣的策劃動作。如果你的計畫是「直接鑽入、開始寫碼」，那也可以（當你對問題的了解已經十分透徹時，這樣做就很適合）。至少請你同意，這也是一種計畫。

你或許得在這個階段勾勒出某些額外的必要程序結構（process structure），但非全貌。某些程式員喜歡用所謂的「渡假模式」來工作，這很可以理解。在這種模式中，他們不會提出開發過程中的任何結構，「當它該被完成的時候，它就會被完成」。這麼做短期間內可能會有吸引力，但我發現，爲整個路程訂定一些里程碑，能夠幫助人們聚集工作的焦點，並在一個個里程碑之間激起對工作的努力，而不至於只是承擔那「完成專案」的唯一目標。此外，這樣也能夠將專案切割爲許多易於解決的小型目標，並且似乎也比較不會感到艱困（多一個里程碑，就多了更多的慶祝機會）。

當我著手於故事結構的研究時（我希望有朝一日我能寫本小說），一開始我相當排斥所謂的結構。我認爲只要自然地讓文字流瀉於紙面，就可以寫出最好的作品。但我很快意識到，當我書寫和電腦相關的事情時，結構對我來說十分清楚，不需特意思考。但我還是將我的工作結構化了，儘管只在某種半意識狀態。是的，即便你認爲你的計畫很簡單，只不過是直接開始寫碼，你仍然得以某種方式經歷接下來的幾個階段，詢問並回答幾個問題。

任務陳述（The mission statement）

不論你要建立的系統有多複雜，都會有其根本目的；它所處的行業、它所要滿足的基本需求。如果你觀看的角度可以穿越使用者界面、與硬體相關或與系統相關的細節、演算法、效率等等，那麼你就可以找出其本質核心所在，那個單純而直觀的本質。就像好萊塢電影所謂的「中心思想（high concept）」一樣，你可以用一個或兩個句子加以描述。如此純粹的描述就是一個起點。

「中心思想」極其重要，因爲它設定了案子的基調，是任務的陳述。你不需要一開始就精確捕捉其真髓（因爲在此事明朗化之前，你可能已經處於稍後數個階段中了），但是請持續努力，直到感覺正確爲止。以飛航流量管制系統爲例，一開始你可能將「中心思想」鎖定在你所建造的系統本身：「塔台程式會記錄飛機行蹤」。但是當你將系統縮小到規模極小的機場時，或許只剩下飛航管制員，或甚至根本沒有人來處理這件事。這裡有一種更實用的模式，便是不關注欲建立之解法本身，而將重點放在問題的描述：「飛機抵達、卸客、檢修、讓顧客登機、啓程」。

階段 1：建立什麼？

Phase 1: What are we making?

前一世代的程式設計（所謂程序式設計，*procedural design*）將此一階段稱爲「產生需求分析和系統規格」。這正是讓人迷途之處；光是那些令人望而生畏的陳述文件，就足以構成一個大型專案了。這些文件的用意是好的；需求分析所做的是：「列舉諸般指導方針，使我們知道何時完成工

作，以及客戶需求滿足的時間點所在」。系統規格則道出：「為了滿足需求，程式做了哪些事情（但並不討論如何完成）」。需求分析就是你和客戶之間的契約（即使該客戶和你服務於同一公司，或甚至它只是某些物件或系統），系統規格則處於探索問題本身的最頂端，在某種程度上，它是對「問題能否被解決、需耗費多少時間」兩個問題的探究。由於二者都需要取得人際共識（而且通常隨著時間改變），所以我認為它們愈少愈好 — 最理想的方式就是採用表列方式和基本圖示，以節省時間。你可能會受到其他束縛，不得不將它擴展為更大的文件，但是請讓最初的文件保持小而精簡。只要幾次腦力激盪會議，配合一位有能力動態描述的主持人，就可搞定這樣的文件。這麼做的目的不僅在於徵求每個人的意見，也是希望促進團隊中協同一致的意見。或許最重要的是，這可以激起大伙兒對專案的熱情。

在這個階段中，將焦點停駐在欲努力達成的目標核心上，有其必要性：決定系統應該達成什麼目標。對此，最有價值的工具是所謂的 **use cases**（使用案例）的聚合。Use cases 會找出系統的關鍵特色，這些特色可以揭示某些即將動用的根本類別（fundamental classes）。此類本質性的描述式解答，可回答如下的問題[9]：

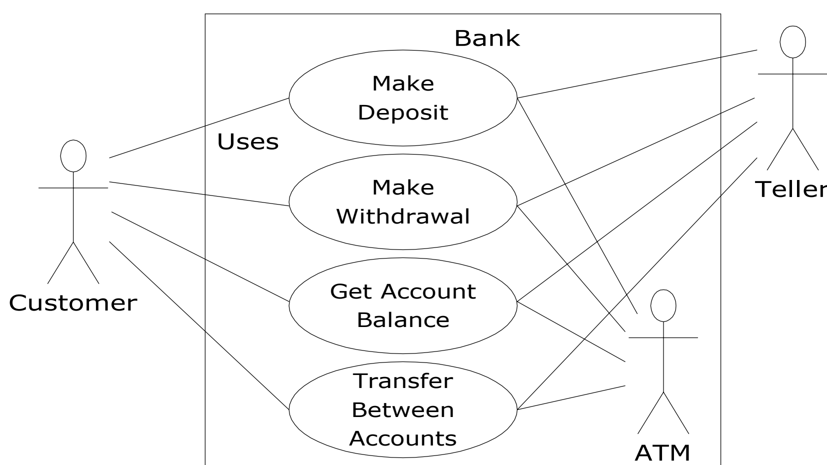
- 誰將使用本系統？
- 系統參與者會透過此系統做些什麼事情？
- 參與者如何透過此系統做到該件事情？
- 如果其他某人正在做這件事，這件事情會以什麼其他方式運作？或問「同一參與者是否具有另一個不同目標？」（顯示出變異情形）
- 透過此系統來進行該件事情時，可能會有哪些問題？（顯示出異常情況）

舉例而言，如果你設計的是自動櫃員系統，針對系統功能特定概況而得的 use cases，便能夠描述自動櫃員系統在每種可能情境下的行為。這些情境都被稱為「腳本（scenario）」，use cases 可被視為腳本的集合體。你可

[9] 感謝 James H Jarrett 協助。

以將某個腳本想像成一個以詢問「如果…的話，系統會怎麼運作？」爲起始的問題。例如「如果某客戶在最近 24 小時內存入一張支票，該支票尚未過戶，帳戶中沒有足夠額度可供提領，此時自動櫃員系統如何處理？」

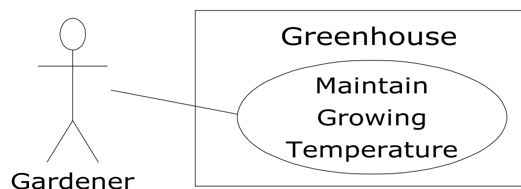
use case diagrams（使用案例圖）刻意設計得十分單純，避免你過早陷於系統實作細節之中：



上圖的每一個棒狀小人，都代表某個「參與者（actor）」，可以是真人，也可以是某種形式的代理人（agent）。甚至可能是其他電腦系統，例如本例的 ATM（自動提款機）一樣。方格代表系統的邊界，橢圓代表 use case，那是系統所能執行的有用工作的描述語句。介於參與者和 use cases 之間的直線，代表兩者的互動。

只要系統對使用者而言，長相的確如此，那麼無論系統實際的實作方式爲何，都不會帶來影響。

即使底層系統十分複雜，use case 也沒有必要極度複雜。其目的只是要顯示使用者所看到的系統的一面。例如：



use cases 會藉著「找出使用者在系統中可能存在的所有互動情形」而產生需求規格。試著找出系統中完整的一組 use cases，一旦完成，系統功能核心便在你的掌握之中。專注於 use cases 的好處是，它能讓你回歸本質面，讓你免於陷入那些「無法對工作的達成產生關鍵影響」的因素。也就是說，如果你擁有了一組完整的 use cases，你便有能力描述你的系統，並有能力進展到下一個階段。或許第一次嘗試時你無法徹底理解，但是不打緊，所有事物都會即時顯現自己。再說，如果你想要在這個時間點上得到一份完美的系統規格，最後不免陷入膠著。

如果你真的陷入膠著，可以使用粗略的近似工具，來發動這個階段的工作：以些許數小段文字來描述系統，同時找尋其中的名詞和動詞。名詞可用以提示參與者、use case 的環境（context，如上例之大廳）、或是在此 use case 中被操作的人工製品。動詞可用以提示參與者和 use case 之間的互動關係，並指出 use case 中的諸多步驟。你當然也會發現，在設計階段中，名詞和動詞還可以對應至物件及訊息（但請注意，use case 描述的是子系統之間的互動關係，因此「名詞和動詞」這種技巧僅能做為腦力激盪工具，無法用來產生 use case）[10]。

介於 use case 和參與者之間的邊界，可以指出使用者介面（UI）的存在，但不定義任何使用者介面。關於使用者介面的定義及建造過程，請參考 Larry Constantine 和 Lucy Lockwood 合著，1999 年由 Addison-Wesley Longman 出版的《*Software for Use*》一書，或是親訪 *www.ForUse.com* 網站。

---

[10] 關於 use case，在 Schneider 與 Winters 合著的《*Applying Use Cases*》（Addison-Wesley 1998）以及 Rosenberg 所著的《*Use Case Driven Object Modeling with UML*》（Addison-Wesley 1999）兩本書中，有更詳盡的解說。

雖然這是一種魔術，但是在這個時間點上制定一些基本時程規畫，還是挺重要的。現在，你已對所要發展的東西有一些概括認識，因此，對於所需花費的時間，你或許能夠捕捉到一些想法。許多影響因素開始在此處發酵。如果你估計的時程過長，公司可能決定不開發（如此便能更合理地將資源用於它處 – 這是一件「好事」）。也許某位經理已經評估過此專案所需的時程，並試著影響你的評估。喔，最好一開始就對時程保持誠實的態度，並儘早處理這個燙手山芋。許多人嘗試建立精確的時程評估技巧，但或許最好的方法就是仰賴你自己的經驗和直覺。先憑直覺來估計所需時間，再將它乘以兩倍，最後再加上百分之十。你的直覺或許正確，讓你可以及時完成專案。乘上兩倍是爲了把事情做得更像樣，最後的百分之十則用在潤飾處理上[11]。不論你怎麼解釋，不論你在呈現如此時程時有著什麼樣的抱怨和運用，最後的結果似乎就是這樣。

階段 2：如何建立？

Phase 2: How will we build it?

在這個階段中，你必須完成設計，其中必須描述 classes 的長相、以及 classes 之間的互動方式。「類別 – 責任 – 協同合作關係（*Class-Responsibility-Collaboration*，**CRC**）」卡片在 classes 及其互動關係識別上，是個極佳技巧。這個工具的價值，部份來自於它的技術單純性：只需一組空白的 3x5 卡片，寫上一些東西，便可開張大吉。每張卡片都代表一個 class，卡片必須寫上：

1. class 的名稱。這個名稱很重要，它得反映出 class 的行爲精髓，令人望而生義。

[11] 對此，我的觀點後來又有所改變。乘以兩倍再加上百分之十，雖然可以得到合理的估計（假設不存在太多 wild-card 因子），但是你仍舊得辛勤工作，才能在時限內完成工作。如果你需要時間以求更精緻的結果，並在過程中得到樂趣，那麼我相信正確的倍數應該是 3 或 4。

2. class 的責任（responsibilities），也就是它應做的事情。通常可以從其成員函式的簡要陳述中獲得（這些函式名稱應該具備清晰的描述力），但也不排除其他註記。如果你想要在這個過程中播種以求收穫，那麼，請從一個懶惰的程式員的角度來檢視問題：「你會期盼怎樣的物件神奇般地出現，一舉解決你的問題？」

3. classes 的「協同合作（collaborations）」關係：哪些 classes 會和這個 class *互動*？*互動*（interact）是個意念廣泛的詞彙，它可能是指聚合關係（aggregation），或只是很簡單地存在某些物件，提供服務給此一 class 的物件。協同合作關係應該考慮到 class 的閱聽者。舉個例子，如果你建立起名爲 **Firecracker**（鞭炮）的 class，那麼誰會觀察（observe）它呢？一個 **Chemist**（化學家）？或是一個 **Spectator**（觀賞者）？前者想知道哪些化學材料可製造鞭炮，後者能夠說出鞭炮爆炸時的顏色和形狀。

譯註：CRC card 是 Kent Beck 和 Ward Cunningham 於 1989 年的 OOPSLA（Object-Oriented Programming Systems, Languages and Applications）學術研討會中，以一篇《*A Laboratory for Teaching Object-Oriented Thinking*》論文提出的。在 Timothy Budd 所著的《*An Introduction to Object-Oriented Programming 2nd Edition*》（Addison-Wesley 1997）中有不錯的應用範例。

你可能會覺得卡片應該大一點，好讓你可以從中取得你想要的所有資訊。但它們的小尺寸其實是刻意的，不僅希望讓你的 class 保持精巧，也希望你不要太早鑽進太深的細節中。如果這般小卡片上的資訊無法滿足你的需求，那就意謂這個 class 太過複雜（如果不是因爲你考慮得太仔細，就是應該產生更多的 classes 來因應）。理想的 class 應該一眼就被了解。CRC card 的出發點，就是要協助你準備第一份設計方案，以利全貌的取得，同時回過頭來修繕原先的設計。

溝通，是 CRC card 帶來的眾多好處之一。最好是不要依賴電腦，在群體中即時完成溝通。每個人都負責數個 classes（一開始可能沒有 classes 名稱，也沒有其他資訊），一次解決一個「腳本」，決定哪些訊息會被發送給哪些不同的物件以符合腳本內容，這樣就彷彿置身於一部生命模擬器中。當你經歷此一過程，你會逐一察覺需要動用哪些 classes、其責任爲何、以及它們之間的合作關係。最終你便能填滿卡片中的欄位。一旦你遍行所有 use cases，一份相當完備的初始設計方案就出爐了。

有一次我和某個開發團隊合作，他們從來沒有以 OOP 方式開發過專案。我站在他們面前，在白板上畫出各種物件。為他們提供設計方案初稿的此次經驗，是我在使用 CRC card 之前，最為成功的一次諮詢經驗。我們討論著物件應當用什麼方式彼此溝通，然後擦掉某些物件，以其他物件替代。其實我只不過是把所有 CRC cards 都放在白板上罷了。該團隊（也就是知道整個專案應該做些什麼事的人）實際產生了設計內容；他們真正「擁有」了這份設計。我只不過是藉著詢問適當問題、嘗試建立假設、接收團隊回饋訊息以修正假設等種種方法，導引整個過程的進行。整個過程最美好的地方莫過於，該團隊並非透過「對某些抽象範例的探討」來學習物件導向設計，而是專注於他們最感趣味的設計之上。

當你準備好一組 CRC cards 時，你可能會想要使用 UML[12] 來為你的設計內容建立更正規的描述。這並非必要，但還是有幫助，特別是當你想要將圖示投射到牆上，好好沉思一番時。UML 之外的另一個替代方案是「物件和其介面的文字性描述語句」，甚至可以是「程式碼本身」[13]（視程式語言而定）。

UML 也針對系統的動態模型，提供額外的圖示符號。當系統或子系統的狀態移轉行為，重要到必須以專屬圖示加以表現時（例如在控制系統中），這些符號就十分有用。一旦資料成了具支配性的影響因素（例如資料庫），你可能也需要描述系統和子系統內的資料結構。

當你完成了物件和其介面的描述，你就會知道階段 2 的工作已告一段落。嗯，其實這只是大部份的物件而已，仍然有一些遺漏掉了，得等到階段 3 才有辦法知道。但這不礙事兒，你只要注意最終是否能夠發掘出所有物件就好了。雖說若能在整個過程中愈早發掘出來最好，但 OOP 提供了足夠的結構，因此即使晚一點才發掘出來也不會造成什麼負面影響。事實上在程式發展的過程中，物件的設計應該發生於五個階段。

---

[12] 對新手而言，我推薦先前提過的《*UML Distilled*, 2nd edition》。

[13] Python（www.python.org）通常被用來做為「可執行的虛擬碼（pseudocode）」。

物件（Object）設計的五個階段

物件的設計並不侷限於程式撰寫時期。事實上物件的設計動作會發生在一連串階段之中。抱持這樣的看法，對你將大有幫助，因爲你不會期盼馬上得到完美的結果，你會意識到，對物件的行爲、外貌長相的理解，是隨著時間而發生的。這樣的觀點能夠應用於不同類型的程式設計上；某個特定型態的樣式（pattern）會在一次又一次與問題對抗的過程中浮現出來（這是《*Thinking in Patterns with Java*》一書的主題，該書可自 *www.BruceEckel.com* 下載）。物件當然也有它們自己的樣式，在了解、使用、重複運用物件的過程中，便會一一顯現出來。

1. 物件的發掘。這個階段發生在程式內部分析期間。透過對以下事項的檢視，包括外在因子和邊界、系統中的元素重複情況、最小概念單元，便可能發掘出物件。如果你已有一組 class libraries，那麼某些物件的存在就很明顯。我們可以透過 classes 之間的共通性來訂定 base classes，繼承關係也許此刻就會明朗，也許還要等到設計後期。

2. 物件的組合：實地發展物件時，你會發現你得增加許多發掘期間未曾出現的新成員。這些物件的內部需求，可能需要其他 classes 的援助。

3. 系統的建構。這個階段中將出現對物件的更多需求。就像學習過程一樣，你的物件會逐步演化。由於得和系統中的其他物件溝通和接駁，因此可能得改變對 classes 的需求，甚至加入新的 classes。舉例來說，你可能會發現需要動用某些諸如 linked list 之類的輔助性 classes，它們包含很少狀態（state），甚至不含任何狀態，只是用來輔助其他 classes 的運作。

4. 系統的延伸。當你將新功能加入系統，你可能會發現，先前的設計無法輕易擴充。因此，你或許可以重新建構系統的部份組成，或許是加入新的 classes，或甚至新的 classes hierarchies。

5. 物件的重複運用。對 class 而言，這無疑是最真實的考驗。如果某人試著將物件重複運用於某個全新情境之中，他也許會找出某些缺點。每當你更動一個 class 以適應更多新程式，那個 class 的一般性原則就會變得更清楚。最後你終於擁有真正可重複運用的 type－ 可以讓大部份物件於特定系統發揮效用。「可重複運用的 types」彼此之間不甚具有共通性，而且，爲了被反覆運用，它們得解決較一般性的問題。

物件（Object）發展的指導原則

上述幾個階段，在你發展 classes 的思路上提供了一些指導原則：

1. 爲特定問題產生一個 class，然後讓它在解決其他問題的同時，漸漸成長茁壯而趨於成熟。
2. 記住，發掘你所需要的 classes 和其介面，是系統設計的主要工作。如果所需的 classes 俱已存在，那麼這個專案再簡單不過了。
3. 別強迫自己一開始就要知道所有事情；耐心地一面前進一面學習。無論如何這都是比較愉快的方式。
4. 開始寫程式；讓某部份先動起來，好讓你可以驗證你的設計，或是找出設計的盲點。別怕被那些和義大利麵條沒兩樣的程序式（procedural style）程式碼逼上死路。classes 能夠切割問題，對於控制無序與混亂很有助益。糟糕的 classes 不會拖累優秀的 classes。
5. 永保單純。物件若能夠保持小而簡潔，並提供明顯易懂的功能，絕對比大而無當、介面繁複的對手來得好。每當面臨抉擇，請使用 Occams 的所謂剃刀法（Razor approach）：在眾多選擇中挑出最單純的一個，因爲最單純的 classes 永遠是最好的。從小而簡潔出發，當你更加了解 class 的介面特性，便可加以擴充。隨著時間推移，我們很難將元素自 classes 中移除。

階段 3：打造核心

Phase 3: Build the core

這個階段進行初次轉換，將原始設計轉換爲「對於可測試之程式碼的主體部份」的編譯與執行動作。這麼做能夠驗證架構的正確性，或找出錯誤所

在。這當然不會只是一個回合而已，這是一個開端，後續一系列步驟將能夠來回反覆地建造出系統，一如你在階段 4 中所見。

我們的目標是找出系統架構的核心，這個核心必須完成 — 不論一開始它是多麼地不完整 — 以便產出可運行的系統。你所產生的正是日後賴以爲根基的框架（framework）。你還會進行首次的系統整合與測試，並給予投資者一些訊息，讓他們知道系統的長相大概是什麼樣子，系統目前的進展又是如何。理想情況下你還會接觸到某些關鍵性風險。你或許也會察覺出原始架構中可更動、可改善的地方，這些都是「不做就學不到」的東西。

系統建立過程中，包含對系統進行真實的檢查，也就是依據需求分析和系統規格（不論以何種形式存在）進行測試。請確認你的測試動作對於需求和 use cases 皆進行了檢驗。系統的核心一旦穩固，也就等於做好了準備，可以繼續往前進，並加入更多功能。

階段 4：use cases 的迭代

Phase 4: Iterate the use cases

一旦核心架構能夠運作，你所加入的每一個功能組，本身都可視爲一個小型專案。你會在所謂「迭代（iteration）」過程中將功能組加入。這個迭代過程，只佔整個發展期相當小的一段時間。

> 譯註：迭代（iteration）是指一種反覆往返的概念。這裡的迭代過程中，我們發展一個新功能，把它整合到先前系統，加以測試。接著再發展另一個新功能、整合、測試…。如此不斷地反覆往返，系統益形增長。

一次迭代過程有多大？理想情形下每個迭代過程持續進行一至三週（隨著程式語言而有不同）。這個過程的最後，你會有一個整合妥當、經過測試的系統，其功能較先前版本更豐富了。特別有趣的是迭代過程的歸依所在：單一 use case。每個 use case 都是一整套相關功能，這些功能是你想要在單一迭代過程中加入系統的。這不僅讓你對 use case 的涵蓋範圍有了更好的認識，也讓你得以印證 use cases 的觀念。即使在分析和設計之後，這個觀念也不應該被丟棄，因爲在軟體發展過程中，無論哪一個階段，它都是最根本的單元。

當你完成你所設定的全部功能，或是外部期限已到，而顧客也對現有版本感到滿意，便是上述迭代過程停止的時刻。（請記住軟體是一種「預訂」行業。）由於整個過程是反覆往返的，所以你有很多出貨機會，而非只能在單一點上；如果你的專案採行開放源碼（open-source）策略，在這種反覆往返、高度倚賴回饋訊息的環境中，能夠如魚得水地獲得成功。

迭代式發展過程，因爲許多因素而顯得彌足珍貴。帶有重大影響的風險可以被提早察覺並加以解決，顧客有足夠機會改變他們的想法，程式設計者可以獲得更高的成就感，也可以更精準地掌控專案的進行。除此之外還有個好處，那就是對投資者的訊息回饋，讓他們可以看到產品中的每一樣事物的當前狀態。這麼做或許可以減少（甚至完全去除）那些叫人傷透腦筋的狀態回報會議，並能夠堅定投資者的信心，增加來自投資者的協助。

階段 5：演化

Phase 5: Evolution

這個階段相當於傳統發展週期中的「維護（*maintain*）」階段。其中涵蓋所有零零總總的事情，包括「讓它可以像一開始就被期望的那樣」到「加一些客戶忘了提到的功能」，甚至更傳統的「修正浮上檯面的臭蟲」和「隨著需求增加，加入一些新功能」，都算在裡頭。許多錯誤的認知都被附加於「維護」一詞，使它承擔了品質上的某些欺瞞行爲。部份原因是這個詞彙讓人認爲，你的確建立了一個原始程式，而你需要做的便是爲它更換零件、上點機油、並避免生鏽。或許我們應該選一個更好的詞彙來描述實際上發生的事情。

我選擇以「演化（*evolution*）」來替代「維護」[14]。其意思是，由於無法在第一回合就讓每一件事正確無誤，所以我們需要一些迴旋空間，一面學習一面回頭修正。隨著你的學習，也隨著對問題本質更加透徹的了解，需要更正的地方或許不少。如果你能不斷演化，直到完全正確，那麼你所展

---

[14] Martin Fowler 的《*Refactoring: improving the design of existing code*》（Addison-Wesley 1999）一書（完全以 Java 爲範例），至少涵蓋了一個以上的「演化」面向。

示的優雅便能取得成功 — 不論是短期或長期。演化，是讓你的程式從「好」到達「了不起」的關鍵，也是那些「第一回合無法真正了解」之問題變得清楚的關鍵。它同時也是你的 classes 從「僅適用於單一專案」演化爲「足堪重複運用」的關鍵。

「事事正確無誤」所指的，並非單只是讓程式根據客戶需求和 use cases 來運作。它同時也意謂，程式碼的內部結構對你來說饒富意義，感覺上服服貼貼地組合在一起，沒有難搞的語法、過大的物件、也沒有任何醜陋的代碼。此外，一旦程式結構面臨更動（那是生命週期中無法逃避的），要如何生存下去，你得有自己的看法。如何使這些更動可以做得輕鬆、做得乾淨俐落，更要有自己的判斷。這不是一朝一夕的功夫。你不僅得了解自己所建的系統，更得了解程式如何演化（即我所稱的「改變的向度（vector of change）」）。幸運的是，物件導向程式語言特別擅長支援此類「持續性修改動作」，是的，由物件所形成的邊界，會主動保持結構免於毀壞。物件導向（OO）程式語言讓你可以進行更動，而你的程式碼不致於處處崩裂（對程序性程式來說「更動」可能是很嚴重的事）。事實上，提供「演化機制」是 OOP 最爲重要的優勢。

有了演化機制，你可以先建立某些看起來近似心中構思的東西，然後把它拿來和設定需求相比較，看看哪些地方不合要求。然後回過頭去修正、重新設計、重新實作程式裡頭尚未正確的部份[15]。想出正確解法之前，你或許得花上好幾次功夫才能解決整個問題，或問題的某一部份。（研究所謂設計樣式（*Design Patterns*），對此將有助益。你可以在《*Thinking in Patterns with Java*》書中找到更多參考資料，此書可自網站 *www.BruceEckel.com* 下載。）

---

[15] 這有點像是所謂的「快速建立原型（rapid prototyping）」。在這種方式下，你可以用快速但相對不那麼乾淨的方式建立一個原型系統，藉此獲得對此系統的了解。然後再將原型系統丟棄，好好地重新打造。「快速建立原型」的問題在於，人們往往捨不得丟棄原型系統，而想以它爲基礎繼續開發。但是你得注意，由於其中結合了程序式編程方法（procedural programming）中「結構不足」的缺點，往往導致混亂，帶來高昂的維護成本。

演化也會發生於你正在建立系統的時候。當你檢查系統是否符合需求，卻發現它並非你想要的東西時，就需要演化了。當你的系統正在運作，然後你才發現其實你要解決的是另一個截然不同的問題，這也需要演化。如果你認爲這種形式的演化正在發生，那麼你應該歸功於自己，因爲你儘快完成了第一版本，才有辦法判斷它是否的確是你想要的東西。

最該銘記於心的重要事情是，預設情形下，如果你修改了某個 class，其 super-classes 和 sub-classes 都應該仍然正常運作。你沒有必要恐懼修改（尤其是當你已經有了一組測試單元，用來檢驗你所做的修改是否正確時，更無需害怕）。修改不必然會帶來毀壞，不過，任何修改的結果都應該被限制在 subclasses 和（或）你所修改之 class 的特定合作者內。

取得成功

當然，你不會在尚未將規畫仔細繪製下來之前，冒然地動手蓋起房子。如果你蓋的只是露天平臺，或者只是狗屋，你的規畫也許不會煞費苦心，但你仍然可能想要稍微描繪個草圖，爲自己領路。軟體的發展已經走到了極端。長久以來，人們並沒有好好地在他們的發展過程中進行結構性規畫，於是大型專案接二連三地失敗。爲了解決這種問題，我們終結了那些瞄準大型專案、有著令人望而生畏的繁多結構與細節事務的方法論。這些方法論令人畏於使用 － 看起來就像會耗掉你的全部時間於文件撰寫上，使你抽不出時間來寫點程式（這的確常常發生）。我希望我所指出的是一條中庸之道 － 一把可移動量度的尺。儘情使用符合你自身需求（以及個性）的方法吧。不論你選擇的方式有多麼簡單，比起毫無計畫而言，「某種」形式的計畫終究還是可以爲你的專案帶來極大的改善。記住，根據統計，超過百分之五十（有些評估甚至認爲是百分之七十）的專案以失敗收場。

透過對計畫的遵循（寧可選擇既簡單又簡短者），並在開始撰碼之前先好好設計結構，你便會發現，所有的事情，相較於「馬上潛心鑽研、開始亂搞一番」，是那麼輕易地可以兜在一起，毫不費力。你會得到無上的滿足。我的經驗是，找出優雅的解決方案，能夠得到極深的愉悅，這種深度愉悅的層次與過去的經驗完全不同；更像是一種藝術，而不單只是技術。優雅的方法不只是無聊的消遣，它們總是能夠成功。它們不但讓你的程式

更易於建立、易於除錯，也更易於理解和維護。而這正是經濟價值之所在啊。

Extreme programming（XP）

從唸研究所開始，我便斷斷續續研讀過不少分析和設計技巧。*Extreme Programming*（**XP**）是我見過最激進、最讓人愉快的一種觀念。在 Kent Beck 所著的《*Extreme Programming Explained*》（Addison-Wesley, 2000）書以及 *www.xprogramming.com* 網站上，都有對此觀念的闡述。

XP 是程式設計工作上的一種哲學，也是一組實踐準則。其中某些準則已反映於晚近問世的其他方法論中。XP 最重要、最獨特的貢獻，就我看來，便是「測試優先（write tests first）」和「搭檔設計（pair programming）」兩者。雖然 Beck 強烈主張完整過程，但他也指出，即使只採用上述兩條實作準則，還是可以大幅提高開發工作的生產力和穩定度。

測試優先

Write tests first

傳統上，測試往往被歸爲整個專案的最末枝節，在「每件事情都正常運作」之後才被考慮，其價值只是爲了「確認真的沒有問題」。這種想法使得大家潛意識裡把測試視爲一件低優先權的工作。專門從事測試的人沒有獲得足夠的地位，而且常常被隔離在地下室，遠離那些所謂「真正的程式研發人員」。測試團隊則回敬說：『我們就像穿著黑袍，把東西弄壞了就樂不可支。』老實說，當我找出編譯器的瑕疵時也有相同的感覺。

XP 讓測試工作和程式撰寫平起平坐（甚至有更高的優先權），這種想法是一種革命性的轉變。事實上你得在撰寫待測程式碼之前，先寫好測試樣例，而後這些測試樣例便永遠與程式碼同在。每次進行專案整合動作時（那很頻繁，甚至可能一天兩次），所有測試樣例都必須能夠成功執行。

「測試優先」可以產生兩個極爲重要的影響。

首先，這麼做能夠強迫人們把 class 的介面定義清楚。當人們試著進行系統設計時，我常常建議他們「幻想有個能夠解決特定問題的完美 class」做爲可用工具。XP 的測試策略又更向前跨了一步，它明確指出 class 對其使用者而言應該具備怎樣的外貌，同時也明確指出 class 必須有的行爲模式。直截了當，毫不含糊。你可以寫下一堆文件、或是產生所有想要的圖示，來描述 class 應有的行爲模式及其外觀長相，但沒有什麼東西比一組測試來得更爲真實。前者像一紙清單，後者卻是一紙契約，一紙由編譯器與程式共同遵守的契約。很難有其他東西能夠比一組測試樣例更具體地描述一個 class 了。

產生測試樣例時，你被迫對 class 進行完整而詳細的考量，並常常因此發現一些需要加入的功能，這些功能也許可能在「以 UML 圖、CRC 卡、use cases 等方式做爲思考進行路線」的過程中漏失。

「測試優先」的第二個重大影響是，你得在每次做出一份軟體成品時，都將所有的測試樣例執行一遍。這麼做可以涵蓋整個測試環節的另一半，也就是編譯器所主宰的部份。如果你從這個角度來看程式語言的發展，你會看到，技術面上最實際的改善其實是以測試爲中心：組合語言只檢查語法正確與否，C 卻加進了某些對語義的限制，避免型別（types）被誤用。OOP 語言加進的語義限制更多 — 如果你把這些限制視爲某種形式的測試的話。「資料型別是否被妥善運用」和「函式是否被正確呼叫」都是由編譯器和執行期系統所進行的測試。我們已經看到了這些測試內建於程式語言後的結果：人們能夠撰寫更複雜的系統，並讓它們運作無誤，但所花費的時間和力氣卻更少了。我曾對此苦思許久，現在我意識到，上述動作都是測試：你有可能弄錯一些東西，而內建測試功能的安全網會告訴你，有問題發生，並指出問題所在。

但是「自程式語言本身的設計切入，藉以提供內建測試機能」，所能做的也就這麼多了。某些情形下你得適時介入，補足完整測試的缺口，藉以對你的程式進行整體檢驗。難道你不希望像編譯器不時從旁照料一般地，讓

這些測試樣例一開始就能夠適時派上用場嗎？這也就是你之所以必須「先寫下測試樣例，並在每次做出一份軟體成品後便自動執行測試」的原因。如此一來，你的測試樣例便能成爲程式語言所提供之安全網的延伸。

使用那些愈來愈具威力的程式語言時，我發現一件事情：由於我知道這些語言有能力使我不必耗費時間於程式臭蟲的捕捉，所以我愈來愈敢嘗試一些尖銳的實驗。XP 的測試體系對整個專案來說，所做之事亦同。因爲你知道你的測試樣例，絕對有能力捕捉到你所引入的所有問題，所以你可以在任何必要的時候，進行大幅度修改，不必憂心整個專案陷入混亂狀態。這真是太有威力了。

搭檔設計

Pair programming

我們向來被諄諄教誨，嚴守個人主義。在學校裡頭，成敗都操之在己。與左右鄰居合作，就被視爲「作弊取巧」。媒體，尤其好萊塢電影，也告訴我們，英雄總是反抗無知的服從[16]。搭檔設計（*pair programming*）悖離這種強烈的個人主義。程式員常常被視爲個體存在的典範 – 就像 Larry Constantine 喜歡說的「牛仔撰碼員（cowboy coders）」。XP 背棄這種想法，認爲應該兩個人合用一部工作站合夥寫程式才對。而且寫程式的環境應該是「在一大塊空間裡頭擺一堆工作站」，不應該在其中樹立室內設計師最喜歡的隔板。事實上，Beck 說，如果想皈依 XP，第一件事情應該是拿起螺絲起子、L 形鈑手，把所有造成阻礙的東西通通拆掉[17]。（那麼當然就得有個經理級的人物，有能力移轉來自設備部門的忿怒囉）

---

[16] 雖然有點大美國主義，但是好萊塢的故事的確流傳各地。

[17] 包括（特別是）擴音裝置。我曾經在某家公司任職，這家公司強烈要求，打給所有業務主管的每一通電話，都必須加以廣播。這麼做時常打斷我們的工作。管理者無法想像，擴音裝置這麼「重要」的服務，有多麼讓人受不了。趁著四下無人，我最後剪掉了擴音機的電線。

搭檔設計（*pair programming*）的價值在於，當某個人思考，另一個人就實際撰碼。從事思考的那個人得將整體宏觀牢記於心 － 不僅僅是對手邊的問題徹底了解，同時還得熟記 XP 的諸般實踐準則。舉例來說，兩個人一起工作，就不太可能會有一個人跳出來說：「我可不想先寫測試樣例。」如果一人陷入困境，兩個人就可以交換角色。如果兩人都陷入困境，那麼他們兩人發呆的樣子，可能會被整個工作區中的某個可以幫得上忙的人發現。這樣子搭檔工作，可以讓事情更平順，並依計畫前進。不過，或許更重要的是，這樣子能讓程式設計變得更與人互動，更充滿樂趣。

我已經在我的某些研討會的習題中加入搭檔設計理念，而我發現，這似乎相當程度地改善了每一個人的經驗。

Java 為什麼成功

Java 能夠取得如此的成功，原因在於其目標的設定：爲當今開發人員解決他們所面臨的諸多問題。提高生產力是 Java 的終極目的。生產力來自於許多層面，但是 Java 語言希望從語言層面提供儘可能的協助。Java 的目的在實用；Java 語言的根本考量，就是爲程式員爭取最多的利益。

易於表達、易於理解的系統

Systems are easier to express and understand

被設計用來「與待解問題相稱」的 class，先天上就有較佳的表述能力。這意謂當你撰寫程式碼時，並不使用電腦術語 — 亦即解域（solution space）空間中的術語，而是以題域空間（problem space）中的術語來描述解法。如此一來便能夠以高階觀念來處理問題，而且每一行程式碼可以做的事情就更多了。

易於表達所帶來另一好處便是維護，而維護佔了整個程式生命週期中極大的成本比例（如果那些數據報告可信的話）。如果程式易於理解，便相對地易於維護。這也同時可以降低文件撰寫與維護的成本。

透過程式庫（libraries）發揮最大槓桿效應

開發程式的最快途徑，就是使用現成的東西：程式庫（library）。Java 的主要目標之一，便是讓程式庫的使用更容易。「將程式庫轉換爲新的 data types（classes）便是 Java 達成此一目標的手段。因此，引入程式庫，意謂將新的 types 加到程式語言裡頭。由於 Java 編譯器會留意程式庫被使用的方式 – 保證初始化動作和清除動作確實正確地執行，並確保呼叫函式的方式合乎規矩 – 你可以專注於程式庫的運用，而不必擔心如何製造它們。

錯誤處理

「錯誤處理」在 C 裡頭向來是個聲名狼藉的問題，而且常被忽略 – 只能祈求上天給你好運。如果你所開發的程式很大，很複雜，那麼大概沒有什麼事比得上「程式某處暗藏一個錯誤，我們卻對其發生原因毫無頭緒」還要糟糕吧。Java 的異常處理（exception handling）便是「一旦發生錯誤，保證一定通知你，並讓你採取一些處理動作」的機制。

大型程式設計

對於程式的大小和複雜度，許多傳統語言都存在若干內建限制。以 BASIC 爲例，它對一般等級的問題，有著最棒的快速解決方案。但是當程式的長度超過數頁，或是逾越該語言的標準題域之外，你就會像「在一池黏稠液體中游泳」一樣，動彈不得。你所使用的程式語言何時會失去作用？噢，沒有一條明白的界線。即使存在界線，你還是會忽略它。因爲你總不能說：「我的 BASIC 程式現在太大了，我得重新用 C 寫過！」所以你會試著硬塞幾行進去，藉以增加新功能。就這樣不知不覺付出了額外的成本。

Java 具備大型程式設計能力 – 也就是抹除了小程式和大程式之間的複雜度界線。如果你只是想寫一個 "Hello World" 小程式，當然不需動用 OOP，但是當你需要用到時，這些功能唾手可得。而且編譯器會積極找出臭蟲所在，不論對小程式或大程式，一律如此。

過渡策略

Strategies for transition

如果你大量採用 OOP，那麼你的下一個問題可能是：「我要如何才能夠讓我的上司、同事、部門、同儕們都開始使用物件呢？」想想你，一個獨立的程式員，如何開始學習一個全新的程式語言和一個全新的設計思維？其實你辦到了。首先是訓練和範例教學，然後是試驗性的專案，讓你感受一些基本原理，而不需要做一些頭昏眼花的事情。接下來就是一個「真實世界」中的專案，這個專案真的能夠做一些有用的事情。在此專案過程中，你不斷閱讀、向高手請教、和朋友交換心得，藉以持續進行學習。這就是有經驗的程式員轉換到 Java 跑道時可以採取的方式。當然，想要轉換整個公司，必然會造成一定程度的群體變動，但是請記住，個人所採取的方式，仍舊可以在每一個步驟中派上用場。

實踐準則

想要過渡到 OOP 和 Java，可以參考以下建議的實踐準則：

1. 訓練

第一步便是某種形式的教育。別忘了公司在程式碼上的投資，並且努力不要讓任何事情處於混亂狀態超過六到九個月 – 雖然每個人都對介面運作的方式感到困惑。挑選一小群人來進行教育，這一小群人最好是個性好奇、能夠彼此合作、能夠在學習 Java 的時候彼此奧援的人。

有時候我也會建議另一種作法，一次教育公司裡的所有階層，包括對決策高層的概觀性說明，以及對專案開發者的設計和撰碼課程。這種方式特別適合小型公司進行根本性的移轉，或是對大型公司的部門層次來進行。不過，由於這會帶來較高的成本，所以也有人選擇從專案層次的訓練開始，先做個前導性專案（可能邀請外聘講師），再讓專案中的成員變成公司其他成員的老師。

2. 低風險專案

先以低風險專案做爲試探，並允許錯誤發生。獲得些許經驗之後，便可以將這個前導團隊的成員播種到其他專案去，或把他們視爲 OOP 技術支援幕僚。第一個專案可能無法一蹴而成，所以不應該挑選會對公司產生重大影響的案子。這個專案應該簡單、能自我控制、具啓發性 — 意謂這個專案應該產生某些 classes，對公司其他人開始學習 Java 時帶來一些意義。

3. 向成功借鏡

起跑之前，先找一些有著良好物件導向設計的範例出來閱讀。通常，有相當的機會，別人早已解決了你的問題。就算沒有人恰巧解決你的問題，你也有可能應用所學，修改他人設計的抽象方法，套用在自己的需求上。這也就是設計樣式（design patterns）的一般概念。《*Thinking in Patterns with Java*》一書涵蓋這個主題，你可以自 *www.BruceEckel.com* 下載。

4. 使用既有的程式庫（libraries）

更換跑道至 OOP，經濟上的主要動機便是能夠以 class libraries 的形式，輕鬆地重複運用既有的程式碼（本書廣泛使用的 Java 標準程式庫更是如此）。當你能夠把既有的程式庫當做基礎，建立物件並加運用，便能產生最短的程式開發週期。不過某些程式設計新手可能無法意識到這一點，也可能沒有發現到程式庫的存在，抑或過度迷戀程式語言的魅力，總想自己動手寫一些早已存在的 classes。如果你能夠早點在這個過渡過程中努力找出他人所寫的程式碼，並加以重複運用，就能夠透過 OOP 和 Java，取得最大的成功。

5. 不要以 Java 重寫既有的程式

將那些既有的、函式風格的程式碼，重新以 Java 寫過，對時間而言通常不是最佳運用方式。如果你一定得把它們轉向物件形式，你可以運用 Java 原生介面（Java Native Interface），用以和 C 或 C++ 程式接軌。本書附錄 B 對此介面有所討論。這麼做可以得到漸進的好處，特別是當那些程式碼被選定重複運用時。不過這麼一來，可能你就無法在剛開始的幾個專案中

看到驚人的生產力提升 — 這一點唯有在全新專案中才有可能出現。Java 和 OOP 的出眾，最能夠表現在讓一個專案的概念成真、落實。

管理上的障礙

如果你是一個管理者，你的工作便是爲你的團隊爭取資源、排除所有影響團隊成功的障礙，並試著提供最能帶來生產力、最讓人樂在其中的工作環境，讓你的團隊成爲最有可能展現奇蹟的團隊，達成那些不可能的任務。遷移至 Java 平台所需付出的成本，會落在以下所列的三個範疇內。如果這並不會讓你付出什麼代價，那真是太好了。有些 OOP 替代方案，是專門爲 C 程式員（或其他程序式語言的使用者）所組成的團隊量身提供的。雖然，相較於那些替代方案，遷移至 Java 平台所需付出的代價，顯然低廉多了，但不論再怎麼低廉的成本，依舊不能不花任何成本。此外，在嘗試將 Java 推廣到你的公司之前、在著手進行遷移計畫之前，你得先知道路上可能有的一些絆腳石，才好應付。

起動成本

遷移至 Java 平台，所需的成本，不單只是取得 Java 編譯器而已（Sun Java 編譯器是免費的，所以這談不上是顆絆腳石）。如果你願意投資在訓練上面（可能得請專家爲你的第一個專案提供顧問諮詢），並且找到了可解決你的問題的程式庫並加以購買，而不是試著自己重新開發這些程式庫，那麼你的中長期成本會降至最低。這些都是得花費實際金錢的成本，必須納入專案提案書。此外，可能有一些隱藏成本，來自於「新語言的學習」和「可能導入的新設計環境」所引起的生產力損耗。訓練和顧問諮詢無疑能夠將這些成本降到最低，但是團隊成員也必須克服他們自身在了解新技術時湧現的掙扎。在這個過程中，他們會犯下更多錯誤（這也是一種特色，因爲大家都公認錯誤是通往學習的捷徑），而且生產力降低。儘管如此，面對某些類型的問題，即便大家還只在學習 Java 的過程中，還是有可能比「停留於 C 的使用」帶來更好的生產力（雖然，所犯的錯誤或許更多，每天所完成的程式碼或許更少）。

效能問題

一個常被問到的問題是：「OOP 不會將我的程式變得既肥且慢嗎？」這個問題的答案是：「視情況而定。」Java 所提供的額外安全功能，傳統上是犧牲諸如 C++ 語言所具備的效能而來的。諸如 "hotspot" 之類的技術和編譯器的技術，多數都能夠大幅提高執行速度；此外還有更多致力於效能提升的努力，正在持續進行。

如果你關注的事情集中在「快速建立原型」上面，你可以匆匆將所有組成拼湊起來，略去效能上的考量。如果你使用的是其他廠商提供的程式庫，它們通常都已經由供應商最佳化了，因此在快速開發模式下，效能通常不是問題所在。當你已經擁有一個想要的系統，如果它夠小而且夠快，你的工作就算告一段落。如果不是這樣，你就得開始使用一些效能研判工具（profiling tool）來進行調整，看看能否改寫小部份程式以提高速度。如果不行，就得找找看有沒有什麼修正手法可以施行於底層，使上層程式碼不需要隨之更動。一旦這些方法都束手無策，你才需要更改設計。由於效能在設計中佔了極重要的地位，所以效能必須成為設計良窳的評量依據之一。藉由快速開發方式，提前找出效能的癥結所在，可以帶來好處。

如果你發現某個函式正是效能瓶頸所在，你可以 C/C++ 重寫那個函式，並以 Java 原生函式（*native methods*）與之接軌，這個主題見本書附錄 B。

常見的設計錯誤

一旦你的開發團隊採用 OOP 和 Java，程式員通常會經歷一連串常見的設計錯誤。通常是因為早期專案的設計和實作上沒有從專家那兒取得足夠的回饋訊息。這可能是因為沒有在公司內培養出專家，也可能是因為存在著對固定顧問諮詢的反對聲浪。很容易就可以感受到你是否在整個週期中過早了解 OOP，並走偏了方向。有些事情，對熟悉此語言的人來說平淡無奇，對新手而言卻是十分重大的討論議題。藉由經驗豐富的外聘專家提供訓練與顧問諮詢，可以避免許多傷害。

Java vs. C++?

Java 看起來很像 C++，而且如此成熟，似乎應該有足夠的份量取而代之。不過我開始質問這種思考邏輯。畢竟 C++ 仍然具備了某些 Java 沒有的特色，而且雖然有許多承諾顯示 Java 終有一天會和 C++ 一樣快，甚至更快，甚至我們也看到了一些效能上的持續進展，但是並沒有什麼驚人突破。再者，整個程式開發環境仍然對 C++ 有持續性的關注，所以我不認爲 C++ 會在任何可見的近期內消失。（程式語言似乎永遠隱而不退。在我的某次 Java 中/高階研討會上，Allen Holub 宣稱，目前最被廣泛使用的兩個語言分別是 Rexx 和 COBOL。）

我於是開始思考 Java 的長處，其戰場其實和 C++ 稍有不同。C++ 這個語言並不試著爲某種類型的問題量身打造。一般而言它可融入許多不同的方法，解決種種問題。有些 C++ 工具將程式庫、組件模型（component models）、程式碼產生工具（code-generation tools）結合在一起，藉以解決視窗環境下（例如 Microsoft Windows）及終端用戶相關的應用程式發展問題。但是，Windows 開發市場中最被廣泛使用的工具是什麼？是 Microsoft Visual Basic（VB） — 儘管 VB 所產生的程式碼類型，會在程式長度超過數頁之後就變得難以管理（而且其語法肯定讓人感到困惑）。即便 VB 如此成功、如此受歡迎，在程式語言的設計上，它不是一個好範例。如果能夠同時擁有 VB 的單純和威力，又不會產生一堆難以管理的程式碼，那就真是太好了。這也正是爲什麼我認爲 Java 會大紅大紫的原因：它是「下一個 VB」。你可能會、也可能不會害怕聽到這樣的說法，但是請想想，那麼多的 Java 功能，都是爲了讓程式員更容易解決一些諸如網絡、跨平台使用者介面之類的應用層次上的問題，更別說它還具備了語言本身的設計，允許極大型而具彈性的程式碼開發工作。再加上一點：Java 具備了我所見過的程式語言中最穩固的型別檢查和錯誤處理機制，你可以飛快提升程式設計的生產力。

那麼，你應該在專案中以 Java 替代 C++ 囉？除非是 Web 上的 applet，否則你應該考慮兩個因素。首先，如果你想使用現成的大量 C++ 程式庫（而

且你將從那兒獲取相當多的生產力），或者如果你已經有了現成的 C 或 C++ 程式碼，那麼 Java 可能會減緩你的開發時程，不會帶來速度的提升。

如果你基本上是從頭開始發展所有的程式碼，那麼 Java 特性之中遠勝於 C++ 的「單純性」，便會大幅縮短你的開發時程。有許多傳言（我從一些原本使用 C++ 後來轉投 Java 陣營的開發團隊聽來的），認為可以帶來兩倍的速度提升。如果 Java 的效能弱點對你而言不重要，或如果你可以其他方法加以補償，那麼在純粹的「市場到位時間」考量下，很難不選擇 Java。

最大的問題還是在效能。使用最原始的 Java 解譯器，Java 大概比 C 慢上 20 到 50 倍。這一點隨著時間已有大幅改進，但仍然有著相當巨大的差距。談到電腦，無非就是速度。如果電腦帶給你的服務沒有快上太多，或許你寧願手動完成。（我曾經聽過有人建議，如果你需要較快的執行速度，你可以先用 Java 獲得較短的開發時間，再以某種工具及其所支援的程式庫，將那些 Java 程式碼轉換為 C++。）

使 Java 足堪適用於大多數專案，關鍵來自於執行速度的提升，例如所謂「just-in time（JIT）」編譯器、Sun 推出的「hotspot」技術、乃至於原生碼（native code）編譯器。當然，原生碼編譯器無法吸引那些需要跨平台執行能力的人，但它帶來的執行速度卻足以逼近 C 和 C++。而且以 Java 寫成的程式，要進行跨平台編譯，比起 C 和 C++來說真是簡單太多了。（理論上你只需要重新編譯。不過以前也有一些程式語言做過相同的承諾。）

你可以在本書第一版附錄中找到 Java 和 C++之間的許多比較，以及對 Java 的現實觀察。（本書所附光碟片中有第一版電子書，你也可以從 *www.BruceEckel.com* 下載）

摘要

本章試著讓你體驗一下 OOP（物件導向程式設計）和 Java 中廣泛的議題，包括：OOP 為何如此與眾不同、Java 又為何如此格外出眾、OOP 方

法論的概念。最後更涵蓋了「當你的公司遷移到 OOP 和 Java 平台時，會遇到的種種問題」。

OOP 和 Java 不見得適用於每個人。重要的是評估你自己的需求，並決定 Java 是否能夠在滿足這些需求上得到最佳結果，抑或使用另一套程式設計系統（包括你正在使用的那一套）對你來說比較好。如果你知道你的需求在可預見的未來會非常特殊，而 Java 可能無法滿足某些特定限制，那麼你應該再考察其他方法[18]。即使你最後選擇了 Java，你至少還得了解有那些項目可供選擇，並對於爲什麼選擇這個方向有非常清楚的看法。

你知道，程序式程式（procedural program）看起來是什麼樣子：就是資料的定義和函式的呼叫。想知道此類程式的意義，你得花上一些功夫，從頭到尾檢視那些函式呼叫和低階概念，才能夠在心中建立起一套模型。這也就是爲什麼我們在設計程序式程式時，需要一些中介表達方法，因爲這種程式天生就容易混淆人們的認知，是的，它們用以表達的詞彙，太偏電腦，不像是你解決問題時所用的術語。

在你能夠找到的程序式語言的所有功能之上，Java 又添加了許多新觀念。因此，你自然而然會假設，Java 程式中的 **main()** 遠比 C 裡頭的兄弟複雜許多。關於此點，你會感到十分驚喜。妥善撰寫的 Java 程式，通常遠比同等性質的 C 程式更單純，更容易理解。你只需查看那些「用來表示題域空間中的概念」（而非任何電腦表述方式）的所謂物件，以及那些在物件之間傳遞的訊息（用來代表題域空間中的活動），就可以輕鬆掌握整個 Java 程式。OOP 的最大樂趣在於，面對妥善設計的程式，只需加以閱讀，很容易就可理解程式碼的內容。事實上，通常不會有太多程式碼，因爲許多問題都已經透過「重複運用既有程式庫」的方式解決掉了。

---

[18] 我特別推薦你看看 Python（http://www.Python.org）

2: 萬事萬物皆物件

Everything is an Object

雖然 Java 奠基於 C++之上，兩相比較，Java 卻是個更「純粹」的物件導向程式語言。

C++和 Java 都是混合型程式語言（hybrid language）。但 Java 的設計者認爲這種混合特性在 Java 中不像在 C++中那麼重要。混合型程式語言提供多種程式設計風格；C++之所以爲混合型語言，是爲了回溯相容於 C。C++ 做爲 C 語言的超集（superset），勢必將 C 語言中許多不適合出現於 C++ 的特性一併囊括進來。這些性質使 C++在某些方面顯得過於複雜。

Java 程式語言在設計上，徹底假設使用者僅以物件導向模式來進行程式設計。這麼一來，你便得先轉換自己的思維模式，進入物件導向的世界，才能夠開始使用 Java（除非你的思維模式早已是物件導向形式）。透過這個入門基本功夫，爾後再學習、再使用其他的 OOP 語言，上手的速度就可以更快。本章之中，我們將看到 Java 程式的基本組成，並學到一個十分重要的觀念：在 Java 程式中，萬事萬物皆物件（object），即使 Java 程式本身，也是一個物件。

Reference 是操控物件之鑰

You manipulate objects with references

每個程式語言都有其獨特的資料操作方式。有時候，程式員對於正在處理的資料，必須持續關注其型別（type）究竟爲何。你究竟是直接操作物件，或是透過某種中介形式（例如 C 和 C++的指標），因而必須以某種特殊語法來看待物件呢？

這一切在 Java 中都大大簡化了。所有事物都被視爲物件，單一固定的語法便可通用各處。不過，雖然抽象概念上你可以把所有東西都「視爲」物件，但用以操控物件的識別字，實際上卻只是其 "reference"（引用、參照、指引）而已[1]。物件和 reference 之間的關係，好比電視和遙控器之間的關係一樣。只要你手上握有遙控器，便可以控制電視。當某人想要改變頻道或是將音量關小一點時，你實際上操控的是遙控器，間接透過這個 reference 來改變實物性質。如果你想在房間裡頭走來走去，同時保有對電視的控制，只要隨身帶著遙控器（也就是 reference），不必揹著笨重的電視。

是的，遙控器於電視機之外獨立存在。你可以擁有某個 reference，卻不見得要將它連接至某個物件。如果你想儲存某個字或某個句子，你可以產生一份 **String** reference，像這樣：

```
String s;
```

但由於這麼寫只會產生一個 reference，不會產生實際的物件，因此此刻將訊息傳送給 **s**，執行時期便會發生錯誤。這是因爲 **s** 並未實際連接到任何實物身上（也就是沒有相對可控制的電視）。所以，比較安全的作法就是在每次產生 reference 的同時便加以初始化：

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

---

[1] 這裡或許會引發爭論。有些人認爲「很明確地，這就是指標（pointer）」。但這種說法對底層實作方式進行了某種假設。事實上，Java 的 reference 在語法上比較接近 C++的 reference。本書第一版中我發明了一個應用於此處的新用語 "handle"，因爲 C++ reference 和 Java reference 相較之下，某些地方還是有著頗爲重要的差異。我那時剛離開 C++ 陣營，而且我不想對那些原本已經了解 C++的程式員帶來混淆，那些人應該會是 Java 使用者的大宗。但是在此第二版中，我決定換回最被廣泛使用的 "reference" 一詞。那些從 C++陣營轉換過來的人們，理應更能妥善理解並運用這個詞彙，無礙地游走於兩個語言之間。儘管如此，還是有人不同意這個術語。我曾經讀過的某本書是這麼說的：「宣稱 Java 支援 *pass by reference* 的那些人，根本一派胡言！事實上是徹底的 *pass by value*」。該作者認爲，Java 的物件識別字實際上就是個 object reference，所以絕對沒有任何 *pass by reference* 之情事，每一樣東西都是 *pass by value*。也許有人會在諸如此類的爭辯上咬文嚼字，但我想我的方法可以簡化觀念的釐清，而不傷害到任何東西。唔，那些精明得像律師的語言學專家也許會認爲我在撒謊，但我認爲我所提供的不過是適度抽象化的結果罷了。

上述程式碼用到了 Java 的一個特性：透過雙引號將文字括起來，便可以對字串進行初始化。不過，這是一種比較特殊的初始化方式。通常你得使用更一般化的方式來對物件進行初始化。

所有物件都必須由你建立

You must create all the objects

產生某個 reference 之後，你多半會想把它連接到某個新產生的物件去。通常我們會透過關鍵字 **new** 來做。**new** 的意思是「請給我一個新物件」。所以上述例子中，你也可以寫成：

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

這麼寫對 **s** 來說，不僅代表著「讓我成為一個新的 **String**」，也提供了用以產生 **String** 的原始字串。

String 並非唯一的既存型別。Java 提供了多不勝數的現成型別。然而重要的是，你可以建立自定型別。事實上這是 Java 程式設計中最主要的一環。透過本章接下來的篇幅，你可以學習如何建立自定型別。

儲存在哪裡

程式執行時究竟如何放置物件？其記憶體佈局方式如何？如果能夠了解這一點，會帶來很大幫助。有六個地方可以存放資料：

1. 暫存器（**Registers**）。這是速度最快的儲存場所，因為暫存器位於處理器內部，這一點和其他種類的儲存媒介都不一樣。不過，由於暫存器個數有限，所以編譯器會根據本身需求適當地配置暫存器來使用。作為一個程式員，你不僅無法直接碰觸暫存器，也沒辦法在程式裡頭感覺到暫存器的任何存在跡象。

2. **Stack**（堆疊），位於一般的 RAM（random-access memory，隨機存取記憶體）中。處理器經由其指標（stack pointer）提供直接支援。當程式配置一塊新的記憶體時，stack 指標便往後移；釋放記憶

體時，指標則往前移回。這種方式不僅很快，效率也高，速度僅次於暫存器。由於 Java 編譯器有責任產生「將 stack 指標前後移動」的程式碼，所以它必須能夠完全掌握它所編譯的程式中「存在 stack 裡頭的所有資料的實際大小和存活時間」。如此一來便會限制程式的彈性。由於這個限制，儘管我們可以將物件的 reference 儲存於 stack 內，但卻不能將一般的 Java 物件也置於其上。

3. **Heap**（堆積）。Heap 是一種通用性質的記憶體儲存空間（也存在於 RAM 中），用來置放所有 Java 物件。Heap 勝過 stack 之處在於，編譯器不需知道究竟得從 heap 中配置多少空間，也不需知道從 heap 上配置的空間究竟需要存在多久。因此，自 heap 配置儲存空間可以獲得高度的彈性。每當你需要產生物件，只需在程式碼中使用 **new**，那麼當它執行的時候，便會自 heap 配置空間。當然，你得為這樣的彈性付出代價：從 heap 配置空間，比從 stack 配置（假設你真的可以在 Java 中像 C++一樣地自 stack 產生物件的話），所耗費的時間多了不少。

4. 靜態儲存空間（**Static storage**）。這裡使用「靜態」一詞，指的是「在固定位置上」（也在 RAM 裡頭）。靜態儲存空間存放著「程式執行期間」一直存在的資料。你可以使用關鍵字 **static**，將某個物件內的特定成員設為靜態，但 Java 物件本身絕無可能置於靜態儲存空間中。

5. 常數儲存空間（**Constant storage**）。常數值常常會被直接置於程式碼裡頭。因為常數是不會改變的，所以也是安全的。有時候常數會和外界隔離開來，所以也可以放到唯讀記憶體（read-only memory，ROM）中。

6. **Non-RAM** 儲存空間。如果資料完全存活於程式之外，那麼即使程式不執行，資料也能夠繼續存在，脫離程式的控制。*streamed objects*（串流化物件）和 *persistent objects*（永續性物件）便是主要的兩個例子。在 streamed objects 形式中，物件被轉換為一連串的 bytes。這些 bytes 通常用來傳送到另一部機器。在 persistent objects 的形式中，物件被儲存於磁碟，所以即使程式結束了，這些物件的狀態還能夠繼續保有。此類儲存空間的特點在於，它們能夠

將物件轉換爲可儲存於其他媒介的形式，並在必要的時候將所儲存的資料還原成可儲存於 RAM 中的一般物件。Java 提供了所謂「輕量級永續性（lightweight persistence）」，新版本有可能提供更完善的解決方案。

特例：基本型別（primitive types）

在「物件型別」（object type）之外，還有一種型別（type）應該被特別對待。你可以把這類型別視爲所謂的「基本型別（primitive types）」，它們會在接下來的程式設計過程中屢屢出現。這一類型別之所以應該受到特別待遇，因爲如果透過 **new** 來產生這一類極小、極簡單的變數，會因「**new** 將物件置於 heap 之上」而效率不彰。因此對於此類變數，Java 採取 C/C++ 的方式，也就是不以 **new** 配置其空間，而是產生一種所謂的 "automatic" 變數（不再是 reference 形式），來解決效率問題（譯註：這便混入了 non-OO 語言性質）。此類變數直接存放資料值，並置於 stack，因此在空間的配置和釋放上，效率好很多。

每一種基本型別所佔的空間大小，在 Java 裡頭是確切不變的。它們的大小不會像大多數程式語言那樣「隨著機器的硬體架構而改變」。這是 Java 程式具備高度可攜性的原因之一。

基本型別	大小	最小值	最大值	外覆型別
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**

所有數值型別（numeric types）都帶有正負號。

boolean 型別的容量未有明確定義；其值僅能爲常數值 **true** 或 **false**。

基本型別有所謂的「外覆類別（wrapper classes）」。如果你想在 heap 內產生用以代表該基本型別的非原始物件（nonprimitive object），那麼，外覆類別便可派上用場。例如你可以這麼寫：

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

或是這麼寫：

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

之所以要這麼寫，其原因得稍後章節才能說明清楚。

高精度數值（High-precision numbers）

Java 提供了兩個用來進行高精度計算的 classes：**BigInteger** 和 **BigDecimal**。雖然它們勉強可以視爲外覆類別，但兩者都沒有對應的基本型別。

不過，這兩個 classes 所提供的操作方法，和基本型別所能執行的十分相像。這句話的意思是，所有可在 **int** 或 **float** 身上做的事情，都可施行於 **BigInteger** 和 **BigDecimal** 身上，只不過必須以「函式叫用方式」取代基本型別的運算子（operators）罷了。這麼做會提高複雜度，所以運算速度會比較慢。是的，這裡打的算盤是：以速度換取精度。

BigInteger 所提供的整數支援任意精度。也就是說你可以很精確地表示任意長度的整數數值，不會在運算過程中喪失任何資訊。

BigDecimal 提供任意精度的定點數（fixed-point numbers）；在需要精確金融計算的場合裡，它便可以派上用場。（譯註：所謂定點數 fixed-point numbers，是指小數點位置固定，或稱小數位數固定數。對應的是所謂的浮點數 floating-point numbers。）

如果想知道這兩個 classes 的建構式（constructors）及其呼叫法，請參閱你手邊的線上文件。

Java 中的陣列（array）

幾乎所有程式語言都提供了陣列。在 C/C++中，陣列的使用是一件危險的事情，因爲這兩個語言的陣列其實就是一塊記憶體而已。如果某個程式存取了陣列區塊之外的位址，或在記憶體被初始化前便使用之（這都是程式設計裡頭再常見不過的錯誤），便會導致不可預期的錯誤發生。

Java 的主要目標之一是安全。許多不斷困擾 C/C++程式員的種種問題，Java 都不再重蹈覆轍。Java 保證陣列一定會被初始化，而且程式員對陣列的存取，無法逾越範圍。這種「對存取範圍的檢查」所付出的代價便是：每個陣列得額外多出一點點空間，並且得在執行時期對陣列索引值進行檢查。Java 認爲，這麼做所帶來的安全性和生產力提升，是值得的。

當你產生某個儲存物件的陣列時，真正產生的其實是個儲存 references 的陣列。此一陣列建立之後，其中的每一個 reference 皆會被自動設爲某個特殊值。該值以關鍵字 **null** 表示。當 Java 看到 **null** 值，便將這個 reference 視爲「不指向任何物件」。使用任何 reference 之前，你必須先將某個物件指派給它。如果你使用某個 reference 而其值爲 **null**，便會在執行期發生錯誤。因此，陣列操作上的常犯錯誤，在 Java 中均可避免。

你當然也可以產生一個陣列，用來儲存基本型別。編譯器一樣保證初始化動作的必然進行：這一次它會將陣列所佔的記憶體全部清爲零。

稍後我們還會再討論陣列。

你再也不需要摧毀物件

You never need to destroy an object

大多數程式語言中，變數的壽命（lifetime）觀念，佔據程式設計工作中非常重要的一部份。變數可以存活多久？如果你認爲應該摧毀某個物件，何時才是最佳時機？圍繞在變數壽命上的種種困惑，可能形成滋生臭蟲的溫床。本節將說明 Java 如何大幅簡化這個議題：是的，它爲你做掉了一切工作。

生存空間（Scoping）

大多數程序式（procedural）語言都有所謂「生存空間（*scope*）」的概念。這個概念決定了某一範圍內的所有變數名稱的可視性（visibility）和壽命。在 C、C++、Java 之中，生存空間由兩個成對大括號決定，例如：

```
{
  int x = 12;
  /* only x available */
  {
    int q = 96;
    /* both x & q available */
  }
  /* only x available */
  /* q "out of scope" */
}
```

生存空間內所定義的變數，都只能用於生存空間結束之前。

程式縮排格式，可以讓 Java 程式碼更易於閱讀。由於 Java 是一種自由格式（free-form）的語言，所以不論空白、跳格、換行，都不會影響程式。

記住，即使以下寫法在 C/C++中合法，在 Java 裡頭卻不能這麼做：

```
{
  int x = 12;
  {
    int x = 96; /* illegal */
  }
}
```

編譯器會認為變數 **x** 已經被定義過了。這種在 C/C++中「將較大生存空間中的變數遮蔽起來」的能力，Java 是不提供的。因為 Java 設計者認為，這麼做容易導致對程式的誤解和混淆。

物件的生存空間

Scope of objects

Java 物件所擁有的壽命，和基本型別是不一樣的。當你使用 **new** 來產生一個 Java 物件，即便離開了生存空間，該物件依然存在。因此如果你這麼寫：

```
{
  String s = new String("a string");
} /* end of scope */
```

s 這個 reference 將在生存空間之外消失無蹤。但是，**s** 先前所指的那個 **String** 物件，仍然會繼續佔用記憶體。如果單看上面這段程式碼，無法存取該物件，因爲唯一指向它的那個 reference，已經離開了其生存空間。後繼章節中你會看到，在程式執行過程中，「指向物件」的那些個 references 是如何地被四處傳遞和複製。

經由 **new** 所產生的物件，會在你還需要用到它時，繼續存在。所以許多 C++程式設計上的問題在 Java 中便消失於無形了。在 C++中，最困難的問題似乎肇因於，程式員不想獲得來自語言的幫助，藉以確保他自己在需要使用物件的時候，物件的確能夠供其使用。更重要的是，在 C++裡頭，你得在用完物件之後，確保它們千真萬確地被摧毀掉。

這麼一來便浮現了一個有趣的問題。倘若 Java 讓這些物件繼續存在，究竟是什麼機制使這些物件不會毫無節制地佔用記憶體，進而搞垮你的程式呢？這正是 C++裡頭可能發生的問題，也正是神奇之所在。Java 之中有一種所謂的「垃圾回收器（*garbage collector*）」機制，它會逐一檢視所有透過 **new** 所產生的物件，並在這些物件不再被引用時（譯註：也就是不再有任何 reference 指向它們時），知道這一事實。然後，垃圾回收器便釋放這些物件的記憶體，提供他用。這代表你根本不用操心記憶體回收問題，只要關心物件的產生就好了。所有物件，在你不再需要它們的時候，都會自動消失。這麼一來便消除了所謂的「記憶體洩漏（*memory leak*）」問題。這個問題正是因爲程式員忘了將記憶體釋放掉而引起的。

建立新的資料型別：class

如果一切都是物件，那麼究竟什麼東西用來決定某一類物件的外觀長相和行爲舉措呢？換另一種方式說，究竟是什麼制定了物件的 type 呢？你也許會預期有個關鍵字 **type**，這才符合它的意義。不過，從歷史沿革來看，大多數物件導向程式語言都使用 **class** 這個關鍵字來代表「我即將告訴你，此一新式物件的長相與外觀」。定義新的 class 時，請在關鍵字 **class**（由於出現太過頻繁，此後不再以粗體字表示）之後緊接著新型別的名稱，例如：

```
class ATypeName { /* class body goes here */ }
```

這麼做便能制定新型別，讓你得以透過 **new** 來產生此一類型物件：

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

AtypeName class 的主體部份只有一行註解（成對的 /* */ 所含括的內容即爲註解。本章稍後馬上會討論之），所以你沒辦法透過它來做些什麼事情。事實上你得先爲它定義一些函式（methods），然後才能夠告訴它如何多做一點事（也就是說，你才能夠將有意義的訊息傳送給它）。

資料成員（fields）和函式（methods）

當你定義 class 時（事實上在 Java 裡頭你需要做的事情無非就是：定義 classes、產生物件、將訊息發送給物件），你可以將兩種成員放進去：資料成員（*data members*，有時稱爲欄位，*field*），以及成員函式（*member functions*），後者通常稱爲 *methods*（譯註：本書將以「函式」稱之）。資料成員可以是任何類型的物件，只要你可以透過它的 reference 來和它溝通就行。資料成員也可以是基本型別（也就是不具備 reference 者）。如果資料成員是一個 object reference，那麼你就得在某個名爲建構式（*constructor*）的特殊函式中（第四章討論），爲該 reference 進行初始化動作，藉以將它連接到某個實際物件去（一如先前所述，以 **new** 來執行這個動作）。如果資料成員是基本型別，你可以直接在 class 的定義處直接給定初值。一如你稍後即將看到，reference 其實也可以在定義處進行初始化。

每一個物件都持有用來儲存資料成員的空間；不同的物件彼此之間並不共享資料成員。下面這個 class 擁有一些資料成員：

```
class DataOnly {
  int i;
  float f;
  boolean b;
}
```

這樣的 class 什麼也不能做。不過你還是可以爲它產生物件：

```
DataOnly d = new DataOnly();
```

你可以指定其資料成員的值，但首先你得知道如何取得（參考到）一個 物件的成員。那就是在 object reference 之後緊接著一個句點，然後再接成員名稱：

```
objectReference.member
```

例如：

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

當然啦，你想修改的資料也有可能位於物件所含之其他物件中。如果是這樣，只要用句點符號把它們連接起來就行了，像這樣：

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

上述的 **DataOnly** class 除了儲存資料之外，什麼事也不能做。因爲它根本沒有任何成員函式（亦即 methods）。如果想了解成員函式的運作方式，你得先了解所謂的引數（*arguments*）和回傳值（*return values*）。稍後對此二者將有簡短說明。

基本成員（primitive members）的預設值（default values）

當 class 的某個成員屬於基本型別（primitive type）時，即使你沒有爲它提供初值，Java 仍保證它有一個預設值。下表列出各基本型別的預設值：

基本型別	預設值
boolean	**false**
char	**'\u0000' (null)**
byte	**(byte)0**
short	**(short)0**
int	**0**
long	**0L**
float	**0.0f**
double	**0.0d**

千萬注意，只有當變數身份是「class 內的成員」時，Java 才保證爲該變數提供初值。這能確保隸屬基本型別的那些資料成員，百分之百會有初值（這是 Java 不同於 C++的地方），因而得以減少許多臭蟲發生機率。不過語言所提供的初值對你的程式而言，或許根本牛頭不對馬嘴，甚至可能不合法。所以最好還是養成習慣，明確爲你的變數指定初值。

上述的初值保證，無法套用於區域變數（也就是並非「某個 class 內的成員」）上頭。因此如果在某個函式定義區內你這麼寫：

```
int x;
```

x 便有可能是任意值（就和 C/C++中一樣），不會被自動設爲零。使用 **x** 之前，你得給它一個適當值。如果忘了這麼做，Java 編譯器會在編譯時發出錯誤訊息，告訴你該變數可能尚未初始化。這是 C++不會發生的事情。（許多 C++編譯器會針對未初始化的變數給予警告，但 Java 將這種情況視爲一種錯誤）

函式（Methods），引數（arguments），回傳值（return values）

到目前爲止，*function*（函式）這個術語用來描述某個具有名稱的子程序（subroutine）。Java 廣泛地使用 *method* 一詞，表示「執行某些事情的

方式」。如果你希望繼續使用 function 一詞，也無妨，只不過是遣詞用字上的差異。從現在起，本書將採用 method 而不採用 function。（譯註：雖然如此，但中譯本為求術語之突出，採用「函式」一詞表示 method）

Java 函式決定了某個物件究竟能夠接收什麼樣的訊息。你將在本節學到，定義一個函式其實是很簡單的。函式基本上包括：名稱、引數（arguments）、回傳型別、主體。下面是最基本形式：

```
returnType methodName( /* argument list */ ) {
  /* Method body */
}
```

函式 被呼叫後，其回傳值的型別就是所謂的「回傳型別」。呼叫者希望傳給函式的種種資訊，則化為引數列（argument list）中的型別和名稱。對 class 而言，函式的「名稱加上引數列」組合，必須獨一無二，不能重複。（譯註：有些書籍很嚴謹區分參數 parameters 和引數 arguments 的不同，本書並不如此）

Java 函式僅能做為 class 的一部份。只有透過物件，而且它能夠執行某個函式，你才能呼叫該函式[2]。如果你對著某個物件呼叫了它並不具備的函式，編譯時期就會得到錯誤訊息。如果想要呼叫某個物件的某個函式，只要指定物件名稱，緊接著句點符號，再接續函式名稱和引數列即可，例如 **objectName.methodName(arg1, arg2, arg3)**。假設有個函式 **f()**，並不接收任何引數，回傳型別為 **int**。如果有個名為 **a** 的物件，允許你呼叫它所擁有的 **f()** 函式，那麼你可以這麼寫：

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

其回傳值型別必須相容於 x 的型別。

這種「呼叫函式」的行為，稱為「發送訊息給物件」。上述例子中，**a** 是物件，**f()** 相當於訊息。物件導向程式設計常常被簡單地歸納為「將訊息發送給物件」的方式。

---

[2] 稍後你會學習所謂 **static** method（靜態函式），它們可透過 class（而非物件）被呼叫。

引數列（The argument list）

外界傳給函式的資訊，由引數列指定。就如你所想的一樣，這些資訊和 Java 的其他資訊一樣，都以物件的形式出現。所以引數列中必須指定每一個想要傳入的物件的型別和名稱。Java 之中，所有傳遞物件的場合（包括現在所討論的這個），傳遞的都是物件的 reference[3]，其型別必須正確。如果引數爲 **String**，那麼你傳入的就必須是個 **String** 物件才行。

假設有個函式，接受 **String** 爲其引數。底下是其定義。這份定義必須被置放於 class 定義式中，才能夠被正確編譯：

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

這個函式告訴你，如果要儲存指定的 **String** 物件，需動用多少 bytes。爲了支援 Unicode 字元集，Java 字串的每個字元都是 16 bits 或說兩個 bytes。引數型別是 **String**，名稱是 **s**。當 **s** 被傳入此一函式，它和其他物件就沒有什麼兩樣了（甚至你可以發送訊息給它）。本例呼叫了 **s** 的 **length()**，那是 **String** 提供的眾多函式之一，會回傳字串中的字元數。

你看到了，上述例子使用關鍵字 **return** 做兩件事情。首先，它代表「離開這個函式」，意謂事情做完了。其次，如果執行過程中誕生了回傳值，這個回傳值應該擺在 **return** 之後。本例之中，回傳值係透過 **s.length() * 2** 這個式子獲得。

定義函式時，你可以決定任何你想要回傳的型別。但如果這個函式並不打算回傳任何東西，你應該將回傳型別指定爲 **void**，例如：

```
boolean flag() { return true; }
```

---

[3]先前所提的那些「特殊」資料型別：**boolean**、**char**、**byte**、**short**、**int**、**long**、**float**、**double**，都是這句話的例外。一般來說，傳遞物件，其實就是傳遞物件的 reference。

```
float naturalLogBase() { return 2.718f; }
void nothing() { return; }
void nothing2() {}
```

當回傳型別為 **void** 時，關鍵字 **return** 就只是用來離開函式。我們不必等到執行至函式最末端才離開，可以在任意地點回返。但如果函式的回傳型別並非 **void**，那麼不論自何處離開，編譯器都會要求你一定得回傳適當型別的回傳值。

閱讀至此，你或許會覺得，程式看起來似乎像是許多「帶有函式」之物件的組合。這些函式可以接受其他物件作為引數，也可以發送訊息給其他物件。以上說法的確還有許多需要補充，不過接下來的章節中，我要先告訴你如何在函式中做判斷，以便執行更細膩的動作。對本章而言，「發送訊息」的討論已經夠了。

打造一個 Java 程式

在看到你的第一個 Java 程式之前，還有一些主題是你必須了解的。

名稱的可視性（Name visibility）

「名稱管理」對所有程式語言而言，都是個重要課題。如果你在程式的某個模組中使用了某個名稱，另一位程式員在同一程式的另一個模組中使用同樣的名稱，該如何區分二者才能使它們不牴觸呢？C 裡頭的這個問題格外嚴重，因為程式之中往往充滿許多難以管理的名稱。C++ classes（Java classes 的師法對象）將函式包裝於內，這麼一來就可以和其他 classes 內的同名函式隔離，避免名稱衝突的問題。不過 C++ 仍允許全域資料（global data）和全域函式（global functions）的存在，所以還是有可能發生命名衝突。為了解決這個問題，C++透過了幾個關鍵字，引入所謂的「命名空間（*namesapces*）」概念。

Java 採用一種全新方法，避免上述所有問題。為了讓程式庫內的名稱不致於和其他名稱相混，Java 採用和 Internet 域名（domain names）相似的指定詞，進一步規範所有名稱。事實上，Java 創設者希望你將 Internet 域名反過來寫，確保所有名稱在這個世界上都是獨一無二的。我的域名是

BruceEckel.com，所以我的一些奇奇怪怪的做爲工具之用的程式庫，就命名爲 **com.bruceeckel.utility.foibles**。當你將域名反轉，其中的句點便用來代表子目錄的劃分。（譯註：例如 com.bruceeckel.utility.foibles 便置於 com/bruceeckel/utility/foibles 目錄下。在這裡，作者探討的是 Java 裡頭的 *package* 命名方式，但未明示 *package* 這個字眼，所以特此提醒讀者。）

在 Java 1.0 和 1.1 中，域名最末的 **com**、**edu**、**org**、**net** 等等，按慣例都應該大寫，所以上例應該寫成：**COM.bruceeckel.utility.foibles**。Java 2 發展到半途的時候，發現這麼做會引起一些問題，因此，現在 package 的整個名稱都是小寫了。

上述機制意謂你的檔案都能夠存在於它們自有的命名空間中（譯註：每個 package 都是一個獨一無二的命名空間），而同一個檔案中的每個 class 都得有個獨一無二的識別名稱。這麼一來就不需特意學習其他語言的什麼功能，便能夠解決這個問題。是的，程式語言自動爲你處理掉這個問題。

使用其他組件（components）

當你想在程式中使用事先定義好的 classes 時，編譯器必須知道它們的位置。當然，它們可能位於同一個原始碼檔案中，那就可以直接在這個檔案中呼叫。這種情形下，你只管使用這個 class – 即使它的定義在檔案稍後才出現。Java 解決了「前置參考（forward referencing）」問題，所以你完全不必傷腦筋。

如果 classes 位於其他檔案之中，又該如何？你可能會認爲編譯器應該有足夠的聰明找到那個位置，可惜事實不然。想像你正要使用某個具有特定名稱的 class，但它卻有好幾份定義（假設各不相同）。或者更糟的是，想像你正在撰寫某個程式，開發過程中你將某個新 class 加到程式庫中，因而和某個舊有的 class 發生了命名衝突。

爲了解決這種問題，你得消除所有可能發生的混淆情形。關鍵字 **import** 可以明確告訴 Java 編譯器，你想使用的 class 究竟是哪一個，以便消除所有可能發生的混淆。**import** 能夠告訴編譯器引入哪一個 package – 那是由 classes 組成的一個程式庫。（在其他語言中，程式庫不僅內含

classes，也可以內含函式和資料，但是在 Java 裡頭，所有程式碼都必須寫在 class 內。）

大部份時候，你會使用 Java 標準程式庫內的種種組件。這個程式庫是和編譯器附在一起的。使用這些組件時，你並不需要寫上一長串的反轉域名。舉個例子，你只要這麼寫就行了：

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

這便是告訴編譯器說，你想使用 Java 的 **ArrayList** class。如果你還想使用 **util** 內的其他 classes，又不想逐一宣告，那就只要以 * 號代表即可：

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

一次宣告一大群 classes，比個別匯入（*import*）一個個 classes，常見且方便多了。（譯註：雖然比較方便，卻會影響編譯時間）

關鍵字 static

一般而言，當你設計某個 class 時，其實就是在描述其物件的外觀長相及行爲舉措。除非以 **new** 來產生物件，否則並不存在任何實質的物件。產生物件之際，儲存空間才會配置出來，其函式才可供外界使用。

但是有兩種情況，是上述方式所無法解決的。第一種是，你希望不論產生了多少個物件，或不存在任何物件的情形下，那些特定資料的儲存空間都只有一份。第二種情況是，你希望某個函式不要和 class object 綁在一起。透過關鍵字 **static**，便可以處理這兩種情況。當你將某筆資料成員或某個函式宣告爲 **static**，它就不再被侷限於所屬的 class object 上。所以，即使沒有產生任何 class object，外界還是可以呼叫其 **static** 函式，或是取用其 **static** data。一般情形下，你得產生某個物件，再透過該物件取用其資料和函式。所以，non-**static** 資料/函式必須知道它們隸屬於哪一個物件，才有辦法運作。由於使用 **static** 函式前並不需要先產生任何物件，所以在 **static** 函式中不能「直接」取用 non-**static** 資料/函式。如果只是單純地直接呼叫 non-**static** 函式，而沒有指定某個物件，是行不通的，原因是 non-**static** 資料/函式總是得和特定的物件綑綁在一起。

某些物件導向程式語言，以 class data 和 class methods 兩個詞彙，代表那些不和特定物件有所關聯，「其存在只是爲了 class」的資料和函式。有時候 Java 相關文獻也會使用這兩個詞彙。

只要將關鍵字 **static** 擺在資料成員或函式的定義前，就可以使它們成爲靜態。以下便可以產生一筆 **static** 資料成員，並將它初始化：

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

現在，即使你產生兩個 **StaticTest** 物件，**StaticTest.i** 仍然只有一份。產生出來的那兩個物件會共用同一個 **i**。再看這個：

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

此時 **st1.i** 和 **st2.i** 的值一樣，都是 47，因爲它們都指向同一塊記憶體。

有兩種方法可以取用宣告爲 **static** 的變數。一如上例，你可以透過某個物件來定址，例如 **st2.i**。也可以直接經由其 class 名稱完成參考動作 — 這種作法對 non-**static** 成員是行不通的（但對於 **static** 成員卻比較好，因爲這種寫法可以更強調所參考的對象是個 **static** 成員）。

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

++運算子會將變數值累加 1。經過這個動作，**st1.i** 和 **st2.i** 的值都是 48。

相同的模式可以推廣到 static 函式。你可以透過物件來取用某個 static 函式，和取用其他普通的函式沒什麼兩樣。你也可以經由以下特殊語法取用之：**ClassName.method()**。定義 static 函式的方式和定義 static data member 的方式十分類似：

```
class StaticFun {
  static void incr() { StaticTest.i++; }
}
```

你看到了，**StaticFun incr()** 將 **static** 資料成員 **i** 的值累加一。你可以用一般方式，透過物件來呼叫 **incr()**：

```
StaticFun sf = new StaticFun();
sf.incr();
```

但由於 **incr()** 是 static 函式，所以你也可以直接透過 class 加以呼叫：

```
StaticFun.incr();
```

某個資料成員被宣告爲 **static** 之後，勢必會改變其建立方式（因爲 **static** 資料成員對每個 class 而言都只有一份，而 non-**static** 資料成員則是每個物件各有一份）。對 **static** 函式來說，差別反而沒那麼大。**Static** 函式 的最重要用途之一，就是讓你可以在不建立任何物件的情形下，還可以呼叫之。這一點很重要，稍後我們會看到，我們得透過 **main()** 的定義，做爲程式的執行起點。

就像任何函式一樣，**static** 函式可以產生或使用其型別所衍生的具名物件，所以 **static** 函式常常被拿來當作「牧羊人」的角色，負責看管眾多隸屬同一型別的一整群物件。

初試啼聲 你的第一個 Java 程式

終於要寫一個真正的程式了[4]。此程式一開始會印出一個字串，然後利用 Java 標準程式庫中的 **Date** class 印出日期。請注意其中運用了一種新的註解方式，雙斜線 //，在它之後出現的所有直到行末爲止的內容，都會被編譯器視爲註解。

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

[4]某些開發環境（programming environments）會將程式輸出畫面快速捲過，在你看不清楚任何東西之前，就什麼都沒有了。你可以將下列一小段程式碼加到 **main()**的最末端，使輸出結果在程式結束前暫停一下：

```
    try {
      System.in.read();
    } catch(Exception e) {}
```

這麼做便能凍結輸出結果，直到按下 "Enter" 或其他鍵。以上程式碼所用到的觀念，一直要到本書後段才會出場。所以此刻你可能無法了解它的作爲，但至少它能派上用場。

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

你得將 **import** 敘述句置於每個程式檔的起始處，藉以將該檔案所需要的所有額外的 classes 含括進來。請注意我說「額外的」，因爲有一組 classes 會被自動含括於每個 Java 程式檔中，我說的是 **java.lang**。請開啓你的 Web 瀏覽器，並檢閱 Sun 的公開文件（如果你尚未從 *java.sun.com* 下載這份文件，或尚未以其他方式安裝 Java 文件的話，此其時矣）。你可以在各個 packages 的列表上找到所有 Java 標準程式庫。請點選 **java.lang**，畫面上會出現一份列表，顯示出其中的所有 classes。由於 **java.lang** 會被自動含括於每個 Java 程式檔中，所以這個程式庫內的所有 classes 都不需要做 import 宣告，便可直接運用。不過 **java.lang** 列表中並沒有 **Date** class，這表示你必須匯入（import）另一個程式庫才能夠使用它。如果你不知道某個 class 位於哪一個程式庫內，或是如果你想同時瀏覽所有 classes，你可以在 Java 文件中選擇 "Tree"，然後便可看到隨 Java 而附的所有 classes。請使用瀏覽器的 "find" 功能搜尋 **Date**。如果你照著上述步驟，就可以看到這個 class 列於 **java.util.Date**，現在你知道它屬於 **util** 程式庫了。爲了使用 **Date** class，你可以寫 **import java.util.*** 將它括進來（譯註：或標明 **java.util.Date** 也行）。

現在回到最開始的地方，選擇 **java.lang** 和 **System**，你會看到 **System** class 有許多欄位。如果你再選擇 **out**，就可以看到它其實是個 **static PrintStream** 物件。因爲它是靜態的，所以你不必做任何事情，**out** 物件便已存在，你只管使用便是。我們到底能對 **out** 物件做什麼事，取決於其型別：**PrintStream**。Java 這份文件設計得很方便，你所看到的 **PrintStream** 是個超鏈結（hyperlink），只要點選它，便可看到 **PrintStream** 提供給外界呼叫的所有函式，數量相當多，本書稍後會加以討論。此刻我們只對 **println()** 感興趣，它的實際作用對我們而言是：「印出我要你印在螢幕上的東西，完成後換行」。因此在你撰寫的 Java 程式中，當你想要將某些訊息列印到螢幕上，可使用 **System.out.println("things")** 完成。

class 名稱必須與檔案主檔名相同。你所開發的程式如果和目前這個程式一樣，是個獨立執行的程式，那麼檔案中必須有某個 class，名稱和檔案主檔名相同。否則編譯器會顯示錯誤訊息。那個 class 必須含有一個 **main()**，其標記式（*signature*）必須是：

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

其中的關鍵字 **public**，表示這是一個要公開給外界使用的函式（第五章有更詳細的說明）。傳入 **main()** 的引數，是個 **String** 物件陣列。這個程式並未使用 **args**，但 Java 編譯器會嚴格要求你一定要這麼宣告 **main()**，因為 **args** 被拿來儲存「命令行（command line）引數」。

印出日期的這一行程式碼，相當有趣：

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

看看傳入的引數：首先產生一個 **Date** 物件，然後直接傳給 **println()**。當這個敘述句執行完畢，產生出來的 **Date** 物件再也不會被使用了，因此垃圾回收器（garbage collector）便會在適當時機將這個物件所佔據的空間收回。此一物件的清除事宜，我們一點也不用擔心。

編譯與執行（Compiling and running）

想要編譯並執行這個程式，以及本書的所有其他程式，你必須先將 Java 開發環境安置妥當才行。目前有許多協力廠商或團體推出各種開發環境，本書假設你使用 Sun 釋出的免費 JDK。如果你使用其他開發系統，你可能需要好好調閱其上所附的文件，決定如何編譯和執行。

上網，然後連到 *java.sun.com*。你可以在這個網站找到相關訊息和鏈結，這些資訊和鏈結會引導你下載符合你的硬體平台的 JDK，並加以安裝。

一旦 JDK 安裝妥當，你也設定好你的路徑（path），使電腦能夠找到 **javac** 和 **java** 這兩個可執行檔（譯註：請參考 p.xxxiii 之「Java 環境設定」），請下載本書原始程式碼（也可以在本書所附光碟中找到，或者從 *www.BruceEckel.com* 網站取得）。解壓縮之後會自動依本書章節，建立起不同的子目錄。現在，請移駕至子目錄 c02，鍵入：

```
javac HelloDate.java
```

這一行命令應該不會產生任何回應。如果你發現任何錯誤訊息，便表示你沒有妥當安裝好 JDK，你得回頭檢查問題出在哪裡。

如果你沒有得到任何回應訊息，請接著輸入：

```
java HelloDate
```

然後便能夠看到程式中的訊息和當天的日期被輸出於螢幕上。

以上整個過程同時也是本書每一個程式的編譯和執行過程。你還會看到本書所附的原始碼中，每章都有一個名爲 **makefile** 的檔案，這個檔案是提供給 "make" 指令用的，可以自動造出（build）該章的所有檔案。*www.BruceEckel.com* 網站上的本書相關網頁，可以告訴你更多如何使用 makefile 的詳細資訊。

註解及內嵌式文件

Comments and embedded documentation

Java 提供兩種註解風格。一種是傳統 C 所用的註解風格，C++也承繼了它。此種註解以 /* 爲首，後接的註解內容可能跨越多行，最後以 */ 結尾。請注意，許多程式員喜歡在多行註解中以 * 做爲每行起頭，所以常常可以看到這樣子的寫法：

```
/* This is a comment
*  that continues
*  across lines
*/
```

記住，/* 和 */ 之間的所有內容，都會被編譯器忽略，所以上述寫法和以下寫法沒有什麼兩樣：

```
/* This is a comment that
continues across lines */
```

Java 的第二種註解風格源於 C++。這種註解用於單行，以 // 爲首，直至行末。這種註解十分方便，並且因爲它的易用性而被廣泛使用。有了這種註解方式，就不必在鍵盤上辛苦尋找 / 和 * 的位置，只需連按兩次相同鍵即可，而且不必注意註解符號的成對問題。所以你常會看到這種寫法：

```
// this is a one-line comment
```

寫文件於註解

Comment documentation

Java 語言有一項經過深思熟慮之後才有的設計。Java 設計者不認爲程式碼的撰寫是唯一重要的工作 － 他們認爲說明文件的重要性不亞於程式碼本身。在程式碼說明文件的撰寫上，最難的問題大概就是文件本身的維護了。如果文件和程式碼二者分離，那麼，每次改變程式碼就得一併更動文件，會是一件很麻煩的事。這個問題的解決辦法似乎很單純：讓程式碼和文件鏈結在一起。想要達到這個目的，最簡單的作法就是把它們都擺入同一個檔案中。不過，更精確地說，這種作法需要某種特殊語法，用以標示出「和程式碼融合在一起」的文件形式。此外也需要一個工具，將這些註解文件自程式碼檔案中提解出來，轉換爲實際可用之文件形式。這正是 Java 的作法。

javadoc 就是用來將程式碼內嵌文件提解出來的工具。這個工具使用 Java 編譯器的某些技術，藉以搜尋比對程式檔中的註解。這個工具不僅會解出由特定標籤（tags）所標示的資訊，也會擷取出這些資訊所屬的 class 名稱或函式名稱。如此一來，一份端端正正、合乎法度的文件，便可以在最小力氣下產生出來。

javadoc 的輸出結果是 HTML 檔案，透過 Web 瀏覽器便可閱讀。這個工具讓你只需維護程式檔，便可自動產生出立即可用的文件。有了

javadoc，大家都可以在文件產生的標準上有所依歸。我們也可以期望或甚至要求，所有 Java 程式庫都得提供相關的說明文件。

語法

所有的 javadoc 命令句，都必須置於以 /** 爲首的註解內，並以 */ 作爲結束。Javadoc 的運用有兩種主要型式：內嵌式 HTML 或文件標籤（doc tags）。所謂文件標籤是一種以 @ 符號爲首的命令，這個符號必須置於註解的最前面，也就是扣掉最前面的 * 之後的最前面處。

文件內容一共有三種型式，分別對應於註解位置前的三種元素：class、variable、method。換句話說，class 的註解必須恰好出現在 class 定義式之前；variable（變數）的註解必須恰好出現在變數的定義之前；method（函式）的註解必須恰好出現在函式的定義式之前。以下是個簡單範例：

```
/** A class comment */
public class docTest {
  /** A variable comment */
  public int i;
  /** A method comment */
  public void f() {}
}
```

請註意，javadoc 只會針對 **public** 或 **protected** 成員，進行註解文件的處理。**private** 或所謂的 friendly 成員（請參閱第五章）會被略去，輸出結果中看不到它們（不過你也可以打開 **-private** 選項，使宣告爲 **private** 的成員一併被處理）。這麼做是有道理的，因爲只有宣告爲 **public** 和 **protected** 的成員，才能夠爲外界所取用；用戶端（client）也只能看到這些。至於針對 **class** 所做的註解，都會被處理並輸出。

上述程式碼的輸出結果是個 HTML 檔，其格式就和所有 Java 文件所遵循的標準一樣，對任何使用者而言肯定不陌生，可從其中輕鬆觀看你所產生的 classes 內容。當你逐一輸入上述程式碼，然後透過 javadoc 產生 HTML 文件，最後再以瀏覽器觀看產出結果，你會覺得這麼做非常值得。

內嵌的 HTML

javadoc 能夠將你所設定的 HTML 控制命令，加到它所產生的 HTML 文件中。這個功能讓你可以盡情發揮 HTML 的功能。不過最主要的目的還是讓你用於程式碼的編排，例如：

```
/**
* <pre>
* System.out.println(new Date());
* </pre>
*/
```

你也可以像編修其它網頁一樣，使用 HTML 格式，將你所製作的一般文字描述，加以排列美化：

```
/**
* You can <em>even</em> insert a list:
* <ol>
* <li> Item one
* <li> Item two
* <li> Item three
* </ol>
*/
```

注意，註解文件中每一行最前面的星號和空白字元，會被 javadoc 忽略。javadoc 會將所有內容重新編排過，俾使輸出結果能夠符合標準的文件外觀規範。請千萬不要在內嵌的 HTML 中使用諸如**<h1>**或**<hr>**之類的標題標籤（headings），因為 javadoc 會插入自己的標題標籤，你的標題會對它造成干擾。

不論是 classes、variables、methods 的註解說明，都支援這種內嵌 HTML 的功能。

@see: 參考其他 classes

三種不同型態（classes、variables、methods）的註解說明中，都可以含入**@see** 標籤。這個標籤的功能是讓你得以參考其他 class 的說明文件。javadoc 會自動為**@see** 標籤產生一個 HTML 超鏈結（hyperlinks），鏈結到其他文件。格式如下：

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

這三種格式都會使產出的文件中多出「See Also」超鏈結。javadoc 並不檢查你所提供的超鏈結內容是否存在，所以你得自己注意。

Class（類別）文件所用的標籤

除了內嵌 HTML 和**@see** 標籤之外，針對類別（classes）而做的文件說明，還可以使用其他標籤來標示作者和版本資訊。這些也都可以套用在 *interfaces*（見第八章）身上。

@version

格式如下：

```
@version version-information
```

其中的 **version-information**（版本資訊）可以是你認爲需要加進去的任何重要訊息。不過，只有在執行 javadoc 時將 **-version** 選項打開，上述的版本資訊才會被加到所產生的 HTML 文件內。

@author

格式如下：

```
@author author-information
```

其中的 **author-information**（作者資訊），顧名生義，就是你的名字。也可以包含你的電子郵箱或其他適合加入的資訊。不過，只有在執行 javadoc 時將 **-author** 選項打開，上述的作者資訊才會被加到所產生的 HTML 文件內。

你也可以提供多個 author 標籤，列出所有作者。這些 author 標籤必須連續出現。產出的 HTML 中，所有作者資訊都會被併到同一段。

@since

這個標籤讓你指定程式碼所使用的最早版本。你可以在 HTML Java 文件中，看到這個標籤被用來指出所使用的 JDK 版本。

Variable（變數）文件所用的標籤

變數說明文件中，除了**@see** 標籤外，只能使用內嵌 HTML 的方式。

Method（函式）文件所用的標籤

除了**@see** 標籤和內嵌 HTML 的方式外，針對函式所做的說明文件，還可以使用描述其參數、回傳值、異常（exceptions）等的控制標籤。

@param

格式如下：

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

其中的 **parameter-name**（參數名稱）是位在參數列中的識別字，**description**（描述句）則是純粹文字，可以延續數行，直到遇上新標籤才結束。此一標籤的使用次數不限，通常我們會在撰寫時爲每一個參數提供一份說明。

@return

格式如下：

```
@return description
```

其中的 **description** 用來描述回傳值的意義；可以延續數行。

@throws

本書第 10 章才會討論異常（exception）。簡單地說，那是一種可以被「擲出（throw）」的物件，當函式在執行過程中發生問題，可以擲出異常，告訴外界發生了這個錯誤。當你呼叫某個函式時，雖然最多只可能出

現一個異常，但函式可能產生任意多個不同型別的異常物件，它們都需要加以描述。描述異常用的標籤格式是：

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

其中的 **full-qualified-class-name** 代表某個異常類別（exception class）的完整名稱，必須獨一無二（譯註：所謂完整路徑是指包含 package 名稱）。**description** 則是用來說明在什麼情形下呼叫這個函式，會產生此一型別的異常。

@deprecated

這個標籤標示，此一函式已被更新的功能替換，建議使用者不要再使用它，因爲它可能在不久的將來被移除。如果你在程式中運用被標示爲 **@deprecate**d 的函式，編譯器會發出警告。

文件製作實例

底下這個程式，是先前介紹過的第一個 Java 程式。這次加上了註解命令：

```
//: c02:HelloDate.java
import java.util.*;

/** The first Thinking in Java example program.
 * Displays a string and today's date.
 * @author Bruce Eckel
 * @author www.BruceEckel.com
 * @version 2.0
*/
public class HelloDate {
  /** Sole entry point to class & application
   * @param args array of string arguments
   * @return No return value
   * @exception exceptions No exceptions thrown
  */
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
```

```
} ///:~
```

在第一行中，我使用自己獨特的方式，將 ":" 作爲一個特殊標示，描述此一註解所在之原始碼檔案的名稱。這一行包含了檔案的完整路徑（本例的 **c02** 代表第 2 章），然後是檔案名稱[5]，最後一行也以註解作收，代表這份原始碼已經到了盡頭，此後再無內容。如此一來便能夠自動化地將這些程式碼自本書文字中取出，再以編譯器驗證之。

撰碼風格（Coding style）

Java 對於 classes（類別）的命名有一個不成文規定：名稱的第一個字母大寫。如果名稱之中含有許多個別字，就把這些字併在一塊兒（不以底線連接），每一個被併在一塊兒的字的第一個字母都採大寫。例如：

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

幾乎所有名稱，包括 methods（函式）、fields（資料成員）、object reference，命名方式都遵循上述法則，唯一的例外是，它們的名稱第一個字母採用小寫而非大寫。例如：

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

當然，你應該體認，classes 使用者在撰寫程式時，也得輸入這麼長的名字。所以命名時請多爲使用者著想，別過度使用無意義又冗長的名稱。

Sun 程式庫中的 Java 程式碼，其左大括號和右大括號的擺放方式，和本書的風格一致。

---

[5] 我利用 Python（請參考 www.Python.org）寫出一個工具，能夠根據這份資訊萃取出程式檔，擺放在適當的子目錄下，並產生 makefile。

摘要

你已經從本章中看到撰寫一個簡單 Java 程式所應具備的相關知識。你也已經對 Java 程式語言有了一個概括性的認識，並了解某些 Java 基本觀念。到目前爲止，所有範例都只停留在「做這件事情，然後做這件事情，接著做那件事情…」的形式。如果你希望程式能夠進行一些判斷，像是：「如果做這件事情的結果是紅色，那麼就做那件事情；否則，就做另一件事情…」，這該怎麼進行呢？Java 所提供的此類程式編寫基本行爲，將在下一章說明。

練習

某些經過挑選的題目，其解答置於《*The Thinking in Java Annotated Solution Guide*》電子文件中。僅需小額費用便可自 *www.BruceEckel.com.*網站取得。

1. 請仿照本章的 **HelloDate.java** 範例程式，寫一個單純印出「hello, world」的程式。你所撰寫的 class 只需具備一個函式即可，也就是程式執行時第一個執行起來的 **main()**。請不要忘了將它宣告爲 **static**，並爲它指定引數列 – 即使你不會用到這個引數列。以 **javac** 編譯這個程式，並以 **java** 執行它。如果你的開發環境不是 JDK，請學習如何在你的環境下進行程式的編譯和執行。
2. 找出含有 **ATypeName** 的程式片段，將它變爲一個完整的程式，編譯後執行。
3. 將 **DataOnly** 的程式片段變成一個完整程式，編譯後執行。
4. 修改習題 3 所完成的程式，讓 **DataOnly** 中的資料可以在 **main()** 中被指定，並列印出來。
5. 撰寫某個程式，含入本章所定義的 **storage()**，並呼叫之。
6. 將 **StaticFun** 程式片段，轉變成一個整個的程式，編譯後執行。

7. 撰寫某個程式，使它能夠接受由命令列（command line）傳入的三個引數。爲此，你得對代表命令列引數的 **String**s 陣列進行索引。

8. 將 **AllTheColorsOfTheRainbow** 這個範例改寫成真正程式，編譯後執行。

9. 找出 **HelloDate.java** 的第二個版本，也就是本章用來說明註解文件的那個程式例。請對該檔案執行 **javadoc**，並透過你的瀏覽器觀看產生的結果。

10. 將 **docTest** 儲存爲檔案並編譯之。然後使用 **javadoc** 爲它產生文件。最後，透過你的瀏覽器觀看產生的結果。

11. 在練習 10 中，以內嵌 HTML 的方式加入一個 HTML 項目列表。

12. 使用練習 1 的程式，爲它加上註解文件。使用 **javadoc** 爲此註解文件產生一個 HTML 檔，並以你的瀏覽器觀看產生的結果。

3: 控制程式流程
Controlling Program Flow

和有情眾生一樣，程式也必須處理它自身的世界，並且在執行過程中有所抉擇。

在 Java 裡頭，物件和資料的處理是透過運算子（operators）來達成，而選擇與判斷則倚靠所謂的控制述句（control statements）。Java 繼承自 C++，因此其大多數述句和運算子對 C 和 C++程式員來說都不陌生。Java 並且在某些地方做了改進和簡化。

如果你發現自己在本章內容的理解上感到費力，請確認自己的確看過本書所附的多媒體光碟《*Thinking in C: Foundation for Java and C++*》中的課程內容。光碟內含有聲課程、投影片、習題、解答。這些材料乃是特別量身打造，能教導你儘早學會在 Java 學習過程中必備的 C 語法。

使用 Java 運算子（operators）

運算子接受一個或多個引數（arguments），並產生新值。引數的形式不同於一般函式，但二者所產生的效應一致。有了過去的編程經驗，你應該很容易接受一般的運算子觀念。加法（+）、減法和負號（-）、乘法（*）、除法（/）、以及賦值（=），其運作方式和其他程式語言幾乎沒有什麼兩樣。

所有運算子都會依據運算元（operands）之值來產生新值。此外，運算子也可以改變運算元之值，此乃所謂「副作用（*side effect*）」。這些會更改運算元內容的運算子，最廣泛的用途便是用來產生副作用。不過你應該牢記於心：使用此類運算子所產生的值，和使用其他類運算子所產生的值，方式上並沒有什麼不同。

絕大多數運算子都只能作用於基本型別上。'='、'=='、'!= ' 是例外，它們可作用於任何物件身上，但這種應用頗易令人迷惑（譯註：就我的觀點，我並不認爲這令人迷惑，反而是一種優點）。除此之外，**String** 類別也支援 '+' 和 '+=' 運算子。

優先序（Precedence）

所謂運算子優先序，定義出單一運算式（expression）內同時出現多個運算子時，該運算式的核定（評估、evaluate）方式。Java 對於核定動作的進行順序遵循特定規則，「先乘除，後加減」是最容易記住的一條規則。其他規則很容易被遺忘，所以你應該使用小括號明確指定核定順序。例如：

```
A = X + Y - 2/2 + Z;
```

便和以小括號加以區分的同一述句（如下），有著不同的意義：

```
A = X + (Y - 2)/(2 + Z);
```

賦值、指派（Assignment）

賦值動作是以 `'='` 運算子爲之。賦值（指派）的意義是取得運算子右邊的值（通常稱爲右值 *rvalue*），將該值複製到運算子左邊（通常稱爲左值 *lvalue*）。右值可以是任何常數、變數、或有能力產生數值的算式，左值則必須是個明確的、具名的變數（也就是說，必須有實際儲存空間以儲存某值）。例如你可以將某個常數指派給某個變數（**A = 4;**），但你無法將任何形式的值指派給常數，因爲常數不能做爲左值（你不能寫 **4 = A;**）。

基本型別的賦值動作相當直覺。因爲基本型別儲存的是實際數值，而非 object reference。當你進行基本型別的賦值動作時，會將某值複製到另一個值身上。例如，對基本型別寫下 **A = B**，**B** 的內容便會被複製到 **A**。如果你接著修改 **A** 值，**B** 當然不會被波及。身爲程式員的你，在大多數情況下都會很自然地這麼預期。但是當你操作某個物件時，你所操作的其實是它的 reference。所以當你「將某個物件指派給另一個物件」，實際上是將

其 reference 從某處複製到另一處。這意謂，如果對是將寫下 **C = D** 這樣的式子，會造成 **C** 和 **D** 都指向原先 **D** 所指的物件。以下例子用來說明這個現象。

```
//: c03:Assignment.java
// Assignment with objects is a bit tricky.

class Number {
  int i;
}

public class Assignment {
  public static void main(String[] args) {
    Number n1 = new Number();
    Number n2 = new Number();
    n1.i = 9;
    n2.i = 47;
    System.out.println("1: n1.i: " + n1.i +
      ", n2.i: " + n2.i);
    n1 = n2;
    System.out.println("2: n1.i: " + n1.i +
      ", n2.i: " + n2.i);
    n1.i = 27;
    System.out.println("3: n1.i: " + n1.i +
      ", n2.i: " + n2.i);
  }
} ///:~
```

Number class 十分單純。它的兩個實體（**n1** 和 **n2**）在 **main()** 中產生出來。每個 **Number** 實體內的 **i** 都被賦予不同之值，然後 **n2** 被指派給 **n1**，然後 **n1** 的內容被改變。在許多程式語言中，你會預期 **n1** 和 **n2** 始終都是獨立而互不干擾，但因爲這裡所指派的乃是 reference，所以你看到的輸出結果是：

```
1: n1.i: 9, n2.i: 47
2: n1.i: 47, n2.i: 47
3: n1.i: 27, n2.i: 27
```

更動 **n1** 內容，同時也更動了 **n2** 內容！這是因爲 **n1** 和 **n2** 內含同一個 object reference。原先 **n1** 所儲存的 reference，乃是指向數值爲 9 的物件，而那個 reference 在賦値過程中被覆寫了，實際上也就是遺失掉了；垃圾回收器（garbage collector）會在適當時機清理該 reference 原本所指的那個物件。

上述現象通常被稱爲 *aliasing*（別名），這是 Java 對於物件的基本處理模式。如果這個例子中你不希望發生別名現象，可以改用這種寫法：

```
n1.i = n2.i;
```

這樣的寫法能讓兩個物件依舊保持相互獨立，無需將 **n1** 和 **n2** 繫結至同一物件並因而捨棄另一個。不過，你很快便會了解，直接操作物件內的欄位會導致混亂，同時也和良好的物件導向設計法則背道而馳。這並不是淺顯的課題，所以我把它留給附錄 A，那兒專門討論別名（aliasing）問題。請千萬不要忘記，物件的指派（賦値）動作會帶來令人意想不到的結果。

呼叫函式時的別名（aliasing）問題

當你將物件傳入函式，也會引發別名現象：

```
//: c03:PassObject.java
// Passing objects to methods may not be what
// you're used to.

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';
    System.out.println("1: x.c: " + x.c);
    f(x);
    System.out.println("2: x.c: " + x.c);
  }
```

```
} ///:~
```

在許多程式語言中，**f()** 會在函式範圍之內爲其引數 **Letter y** 製作一份複本。但因爲現在傳入的其實是個 reference，所以這一行：

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

實際上會更動到 **f()** 範圍外的那個原本物件。程式結果說明了這一點：

```
1: x.c: a
2: x.c: z
```

別名（aliasing）所引起的問題及其解決方法，是個很複雜的議題。雖然你必須閱讀附錄 A 的內容才能得到所有解答，但此時你應該知道有這麼一個問題，才能夠處處小心而不至於落入陷阱。

數學運算子（Mathematical operators）

Java 的基本數學運算子和大多數程式語言一樣：加法（+）、減法（-）、除法（/）、乘法（*）、模數（%，用來取得整數相除後的餘數）。整數除法會將所得結果的小數部份截去，不會自動四捨五入。

Java 也使用簡略標記形式，讓某種運算動作和賦值動作同時進行。這種簡略標記法是在運算子之後緊接著等號，適用於 Java 語言中的所有運算子（如果對該運算子而言，這種寫法有意義的話）。例如想要將變數 **x** 加 4 並將結果指派給 **x**，就可以這麼寫：**x += 4**。

下面這個範例說明了數學運算子的使用：

```
//: c03:MathOps.java
// Demonstrates the mathematical operators.
import java.util.*;

public class MathOps {
  // Create a shorthand to save typing:
  static void prt(String s) {
    System.out.println(s);
  }
  // shorthand to print a string and an int:
```

```
  static void pInt(String s, int i) {
    prt(s + " = " + i);
  }
  // shorthand to print a string and a float:
  static void pFlt(String s, float f) {
    prt(s + " = " + f);
  }
  public static void main(String[] args) {
    // Create a random number generator,
    // seeds with current time by default:
    Random rand = new Random();
    int i, j, k;
    // '%' limits maximum value to 99:
    j = rand.nextInt() % 100;
    k = rand.nextInt() % 100;
    pInt("j",j);  pInt("k",k);
    i = j + k; pInt("j + k", i);
    i = j - k; pInt("j - k", i);
    i = k / j; pInt("k / j", i);
    i = k * j; pInt("k * j", i);
    i = k % j; pInt("k % j", i);
    j %= k; pInt("j %= k", j);
    // Floating-point number tests:
    float u,v,w;  // applies to doubles, too
    v = rand.nextFloat();
    w = rand.nextFloat();
    pFlt("v", v); pFlt("w", w);
    u = v + w; pFlt("v + w", u);
    u = v - w; pFlt("v - w", u);
    u = v * w; pFlt("v * w", u);
    u = v / w; pFlt("v / w", u);
    // the following also works for
    // char, byte, short, int, long,
    // and double:
    u += v; pFlt("u += v", u);
    u -= v; pFlt("u -= v", u);
    u *= v; pFlt("u *= v", u);
    u /= v; pFlt("u /= v", u);
  }
} ///:~
```

首先映入眼簾的是一些用於列印的簡單函式：**prt()** 用來列印某個 **String**，**pInt()** 會在列印某個 **String** 之後緊接著印出一個 **int**，**pFlt()** 會在列印 **String** 之後緊接著印出一個 **float**。當然，這些函式最終都會用到 **System.out.println()**。

爲了產生許多數字，此程式首先產生一個 **Random** 物件。產生這個物件時我們並未傳入任何引數，所以 Java 使用執行當時的時間作爲亂數種子。此程式會透過 **Random** 物件呼叫不同的函式：**nextInt()**、**nextLong()**、**nextFloat()**、**nextDouble()**，產生多個不同型別的亂數。

將模數（modulus）運算子 % 作用於亂數產生器所產生的亂數身上，目的是爲了讓隨機亂數的最大值侷限在我們所指定的運算元數值減 1（本例爲 99）範圍內。

一元（Unary）運算子：負號（minus）和正號（plus）

負號（-）和正號（+）都是一元運算子，其符號和二元運算子中的加法和減法相同。編譯器會依據算式的寫法，判斷你想使用的究竟是哪一種。例如以下述句的意義就很明顯：

```
x = -a;
```

編譯器也可以理解以下述句：

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

但是程式閱讀者可能感到困惑，所以這樣子寫更爲明確些：

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

負號運算子會取得其運算元的負值。正號運算子的功能和負號運算子相反 – 其實它沒有產生任何影響。

遞增（increment）和遞減（decrement）

Java 和 C 一樣，充滿著許多能夠帶來便捷的手法，它們能使程式碼更容易完成，有可能使程式碼更易於閱讀，但也有可能造成反效果。

在許多便捷手法中，遞增和遞減運算子是兩個好東西（通常稱爲自動遞增和自動遞減運算子）。遞減運算子的符號是--，意指「減去一個單位」。遞增運算子的符號是++，意指「加上一個單位」。如果 **a** 是個 **int**，那麼 **++a** 便和 **a=a+1** 等價。因此，遞增和遞減運算子會爲運算元產生新值。

遞增和遞減運算子各有兩個版本，通常稱爲前序（*prefix*）版本和後序（*postfix*）版本。前序遞增是指 ++ 運算子出現於變數或算式之前，後序遞增則是指 ++ 運算子出現於變數或算式之後。同樣道理，前序遞減是指 -- 運算子出現在變數或算式之前，後序遞減是指 -- 運算子出現在變數或算式之後。對前序遞增和前序遞減（也就是 **++a** 和 **--a**）而言，會先進行運算然後才指派其值。而後序遞增和後序遞減（也就是 **a++** 和 **a--**）而言，會先指派其值然後才進行運算。下面是個實例：

```
//: c03:AutoInc.java
// Demonstrates the ++ and -- operators.

public class AutoInc {
  public static void main(String[] args) {
    int i = 1;
    prt("i : " + i);
    prt("++i : " + ++i); // Pre-increment
    prt("i++ : " + i++); // Post-increment
    prt("i : " + i);
    prt("--i : " + --i); // Pre-decrement
    prt("i-- : " + i--); // Post-decrement
    prt("i : " + i);
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

輸出結果是：

```
i : 1
++i : 2
i++ : 2
i : 3
--i : 2
```

```
i-- : 2
i : 1
```

你看到了，以前序（prefix）形式爲之，會在運算執行完畢後才擷取其值（變數或算式的值），如果以後序（postfix）形式爲之，會在運算執行之前便先擷取其值以爲它用。所有運算子中，會引起副作用的，除了那些帶有賦值動作的運算子外，就是這些遞增、遞減運算子了。也就是說，這些運算子會改變（而不單單只是使用）運算元的值。

遞增運算子是 C++的名稱源由之一，意思是「超越 C 更進一步」。Bill Joy（Java 的創造者之一）曾經在一場 Java 演講中說過：Java=C++--（C 加加再減減）。這句話的意思是：Java 是「移去累贅、困難部份之後的 C++」，因此是個更爲單純的程式語言。當你逐步閱讀本書，你會發現，Java 的確在許多地方更爲單純，但 Java 卻不比 C++簡單太多。

關係運算子（Relational operators）

關係運算子所產生的結果是 **boolean**。此類運算子會評估兩個運算元之間的關係。如果其關係爲真，運算結果便爲 **true**。如果其關係爲僞，運算結果便爲 **false**。關係運算子有小於（<）、大於（>）、小於等於（<=）、大於等於（>=）、等於（==）、不等於（!=）等共六種。== 和 != 可作用於所有內建型別身上，其他運算子無法作用於 **boolean** 型別。

物件相等性（object equivalence）的測試

關係運算子 == 和 != 也可作用於任何物件身上，但這兩個運算子的意義常常會對初次接觸 Java 的程式員帶來困惑。以下便是個例子：

```
//: c03: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);
  }
} ///:~
```

System.out.println(n1 == n2) 這行算式會印出括號中的比較結果（**boolean** 値）。想當然爾，本例輸出結果當然先是 **true** 而後 **false**，因爲兩個 **Integer** 物件的值是相同的。不過雖然兩個物件的內容相同，其 references 卻不同。由於 `==` 和 `!=` 運算子所比較的是 object references，所以實際輸出結果是 **false** 而後 **true**。這樣的結果當然會令初次接觸 Java 的人們大感驚訝。

如果我們想知道物件的內容是否相等，又該如何？你得使用 **equals()**。任何一個物件（不含那些能夠正常運用 `==` 和 `!=` 的基本型別）都擁有這個函式。以下便是其運用方式：

```
//: c03: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));
  }
} ///:~
```

結果如你所預期，印出 **true**。呃，不過事情並非如此簡單。如果你建立自有的 class，好比這樣：

```
//: c03:EqualsMethod2.java

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));
  }
} ///:~
```

情況再度回到原點：輸出結果是 **false**。這是因為 **equals()** 的預設行為是拿 references 來比較。所以除非你在你的 classes 中覆寫（*override*）**equals()**，否則便得不到你想得到的行為。不幸的是直到第七章你才會學到所謂的覆寫技術。儘管如此，了解 **equals()** 的運作方式，還是可以讓你免於犯下某些錯誤。

Java 標準程式庫中的大多數 classes 都覆寫了 **equals()**，所以它們都會比較物件（而非其 references）的內容是否相等。

邏輯運算子（Logical operators）

邏輯運算子 AND（&&）、OR（||）、NOT（!）都會得到 **boolean** 值。此值究竟是 **true** 或 **false**，取決於引數間的邏輯關係。下面這個例子使用關係運算子和邏輯運算子：

```
//: c03:Bool.java
// Relational and logical operators.
import java.util.*;

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

    // Treating an int as a boolean is
    // not legal Java
//! prt("i && j is " + (i && j));
//! prt("i || j is " + (i || j));
//! prt("!i is " + !i);
```

```
    prt("(i < 10) && (j < 10) is "
       + ((i < 10) && (j < 10)) );
    prt("(i < 10) || (j < 10) is "
       + ((i < 10) || (j < 10)) );
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

你只能將 AND、OR、NOT 施用於 **boolean** 值身上。邏輯算式中的 **boolean** 值無法以 non-**boolean** 值替代，這在 C/C++中卻是可以的。你可以看到上例某個部分由於這個原因而導致失敗，該部分已用 //! 標示起來成爲註解。緊接於其後的算式則使用邏輯運算子（logical operators）來產生 **boolean** 值，然後才將邏輯運算施加於所產生的 **boolean** 值身上。

輸出結果可能是這樣（譯註：由於採用亂數，每次結果可能不盡相同）：

```
i = 85
j = 4
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 true
```

請注意，如果 **boolean** 值被用於某個預期會出現 **String** 的地方，其值會被自動轉換爲適當的文字形式。

你可以在上述程式中以任何 non-**boolean** 基本型別來替換 **int**。不過請務必明白，浮點數的比較是很嚴格的。兩個相差極微的浮點數仍然是不相等的。是的，一個只比零大一點點的數字，依然不能說是零。

遽死式（短路式，Short-circuiting）核定

處理邏輯運算子時，有所謂「*short circuiting*（遽死、短路）」的現象發生。意思是說，當整個算式的值可以被確切判斷出真僞時，算式的評估

（核定）動作便會結束。如此一來，邏輯算式中的某些部份就可能不會被評估到。以下是個例子：

```
//: c03:ShortCircuit.java
// Demonstrates short-circuiting behavior.
// with logical operators.

public class ShortCircuit {
  static boolean test1(int val) {
    System.out.println("test1(" + val + ")");
    System.out.println("result: " + (val < 1));
    return val < 1;
  }
  static boolean test2(int val) {
    System.out.println("test2(" + val + ")");
    System.out.println("result: " + (val < 2));
    return val < 2;
  }
  static boolean test3(int val) {
    System.out.println("test3(" + val + ")");
    System.out.println("result: " + (val < 3));
    return val < 3;
  }
  public static void main(String[] args) {
    if(test1(0) && test2(2) && test3(2))
      System.out.println("expression is true");
    else
      System.out.println("expression is false");
  }
} ///:~
```

每個測試動作都對傳入的引數進行了比較，並回傳 true 或 false。同時也列印訊息，表示該函式正被呼叫。這些測試動作被使用於以下算式中：

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

你可能很自然地認爲所有測試動作都會被執行，但輸出結果卻說明事實並非如此：

```
test1(0)
```

```
result: true
test2(2)
result: false
expression is false
```

第一個測試結果爲 **true**，所以算式評估動作繼續進行。第二個測試結果爲 **false**，這意謂整個算式結果必爲 **false**，那麼還有什麼理由得繼續算式剩餘部份的評估呢？繼續執行無意義的評估，代價可能很昂貴。*short-circuiting* 的存在正是基於這個原因。如果可以不必評估算式中的所有部份，將因此帶來效率的提升。

位元運算子（Bitwise operators）

位元運算子讓你可以操作基本整數型別中的個別位元。位元運算子會在兩個引數的相應位元上執行 boolean 代數運算以求結果。

位元運算子承襲 C 語言的低階定位：你時而需要直接處理硬體，並設定硬體暫存器中的位元。由於 Java 一開始是針對內嵌於電視的 set-top boxes（譯註：或譯爲機上盒，通常與電視相連，提供許多和電視整合的服務）而設計，因此這個低階定位對 Java 來說仍具意義。不過或許你不會太常用到位元運算子。

位元運算子 AND（&）會在兩個輸入位元皆爲 1 時，產生一個輸出位元 1；否則爲 0。位元運算子 OR（|）會在兩個輸入位元中有任何一個爲 1 時，產生一個輸出位元 1；當兩個輸入位元皆爲 0，結果爲 0。位元運算子 EXCLUSIVE OR，或稱 XOR（^），會在兩個輸入位元恰有一個爲 1（但不可同時爲 1）時，產生結果値 1。位元運算子 NOT（~），也稱爲「一的補數（*one's complement*）」運算子，是個一元運算子（其他位元運算子都是二元運算子），僅接受一個引數，它會產生輸入位元的反相：如果輸入位元爲 0，結果就是 1，如果輸入位元爲 1，結果就是 0。

位元運算子和邏輯運算子使用同一套運算符號。爲了加以區分，如果有個助憶法來協助我們，將會大有助益。是的，由於位元很「小」，所以位元運算子僅使用一個字元符號，邏輯運算子使用兩個字元符號。

位元運算子也可以和 = 併用，使運算動作和賦值動作畢其功於一役：&=、|=、^= 都是合法的。至於 ~，由於是一元運算子，無法與 = 合併使用。

Boolean 值被視爲單一位元，所以情況有點不同。你可以在其身上執行 AND、OR、XOR 位元運算，但不能執行 NOT 運算（大概是爲了避免與邏輯運算 NOT 混淆）。對 **boolean** 而言，位元運算子除了不做 short circuit（遽死式、短路式評估）外，和邏輯運算子是相同的。此外，可作用於 **boolean** 身上的位元運算，還包括不含於邏輯運算子中的 XOR 運算。最後一點，**boolean** 值無法用於位移運算（稍後即將說明）。

位移運算子（Shift operators）

位移運算子也用來操作位元，但僅用於基本整數型別身上。左移運算子（<<）會將左運算元向左搬移，搬移的位元個數由右運算元指定（左移後，較低位元會被自動補 0）。帶正負號（signed）的右移運算子（>>）則將左運算元向右搬移，搬移的位元個數由右運算元指定。面對帶正負號的數值，右移動作會採用符號擴展（*sign extension*）措施：如果原值是正數，較高位元便補上 0；如果原值是負數，較高位元便補上 1。此外 Java 還增加了無正負號的右移運算子 >>>，採用所謂的零擴展（*zero extension*）：不論原值是正或負，一律在較高位元處補 0。「無正負號右移運算子」在 C 和 C++中並不存在。

如果你所操作的位移對象是 **char**、**byte**、**short**，位移動作發生之前，其值會先被晉升成 **int**，運算結果會是 **int**。運算子右端所指定的位移個數，僅有較低的 5 個位元有用。這樣可以避免你移動超過 **int** 所具備的位元數（譯註：2 的 5 次方是 32，而 Java 的 **int** 正是 32 位元）。如果你所操作的對象是 **long**，運算結果也會是 **long**，而你所指定的位移個數僅有較低的 6 個位元有用（譯註：因爲 **long** 是 64 位元），這能夠避免你移動的位元數超過 **long** 所具備的位元數。

位移運算也能和等號合併使用（<<= 或 >>= 或 >>>=）。新的左值會是原左值位移了「右值所指定的位元數」後的結果。不過，當「無正負號右移動作」配合「賦值動作」使用時，會有問題：在 **byte** 或 **short** 身上無法

得到正確結果。是的，它們會被先晉升為 **int**，然後進行右移；但是當它們被賦值回去時，其值又會被截去（譯註：超過容量大小的較高位元會被截去）。這種情況下會得到 **-1**。下例即說明這個問題：

```
//: c03:URShift.java
// Test of unsigned right shift.

public class URShift {
  public static void main(String[] args) {
    int i = -1;
    i >>>= 10;
    System.out.println(i);
    long l = -1;
    l >>>= 10;
    System.out.println(l);
    short s = -1;
    s >>>= 10;
    System.out.println(s);
    byte b = -1;
    b >>>= 10;
    System.out.println(b);
    b = -1;
    System.out.println(b>>>10);
  }
} ///:~
```

最末一行的結果並未被指派回 **b**，而被直接印出，所以產生正確的行為。

下面這個範例展示所有和位元運算有關的運算子：

```
//: c03:BitManipulation.java
// Using the bitwise operators.
import java.util.*;

public class BitManipulation {
  public static void main(String[] args) {
    Random rand = new Random();
    int i = rand.nextInt();
    int j = rand.nextInt();
    pBinInt("-1", -1);
    pBinInt("+1", +1);
```

```
    int maxpos = 2147483647;
    pBinInt("maxpos", maxpos);
    int maxneg = -2147483648;
    pBinInt("maxneg", maxneg);
    pBinInt("i", i);
    pBinInt("~i", ~i);
    pBinInt("-i", -i);
    pBinInt("j", j);
    pBinInt("i & j", i & j);
    pBinInt("i | j", i | j);
    pBinInt("i ^ j", i ^ j);
    pBinInt("i << 5", i << 5);
    pBinInt("i >> 5", i >> 5);
    pBinInt("(~i) >> 5", (~i) >> 5);
    pBinInt("i >>> 5", i >>> 5);
    pBinInt("(~i) >>> 5", (~i) >>> 5);

    long l = rand.nextLong();
    long m = rand.nextLong();
    pBinLong("-1L", -1L);
    pBinLong("+1L", +1L);
    long ll = 9223372036854775807L;
    pBinLong("maxpos", ll);
    long lln = -9223372036854775808L;
    pBinLong("maxneg", lln);
    pBinLong("l", l);
    pBinLong("~l", ~l);
    pBinLong("-l", -l);
    pBinLong("m", m);
    pBinLong("l & m", l & m);
    pBinLong("l | m", l | m);
    pBinLong("l ^ m", l ^ m);
    pBinLong("l << 5", l << 5);
    pBinLong("l >> 5", l >> 5);
    pBinLong("(~l) >> 5", (~l) >> 5);
    pBinLong("l >>> 5", l >>> 5);
    pBinLong("(~l) >>> 5", (~l) >>> 5);
  }
  static void pBinInt(String s, int i) {
    System.out.println(
      s + ", int: " + i + ", binary: ");
```

```
    System.out.print("   ");
    for(int j = 31; j >=0; j--)
      if(((1 << j) &  i) != 0)
        System.out.print("1");
      else
        System.out.print("0");
    System.out.println();
  }
  static void pBinLong(String s, long l) {
    System.out.println(
      s + ", long: " + l + ", binary: ");
    System.out.print("   ");
    for(int i = 63; i >=0; i--)
      if(((1L << i) & l) != 0)
        System.out.print("1");
      else
        System.out.print("0");
    System.out.println();
  }
} ///:~
```

最末尾的兩個函式 **pBinInt()** 和 **pBinLong()**分別接受單一 **int** 或 **long** 做爲引數，並以二進位格式搭配說明字串印出。此刻你可以先忽略其實作細節。

請注意，這裡使用 **System.out.print()** 替代 **System.out.println()**。**print()** 並不自動換行，所以我們可以分次輸出單行中的內容。

除了說明位元運算子在 **int** 和 **long** 身上的效果，本例也顯示出 **int** 和 **long** 的最大值、最小值、**+1** 實值、**-1** 實值，讓你清楚看到它們的長相。請注意，最高位元代表正負號：0 代表正值（譯註：包括零），1 代表負值。例中關於 **int** 的部份，輸出結果像這樣：

```
-1, int: -1, binary:
   11111111111111111111111111111111
+1, int: 1, binary:
   00000000000000000000000000000001
maxpos, int: 2147483647, binary:
   01111111111111111111111111111111
maxneg, int: -2147483648, binary:
```

```
   10000000000000000000000000000000
i, int: 59081716, binary:
   00000011100001011000001111110100
~i, int: -59081717, binary:
   11111100011110100111110000001011
-i, int: -59081716, binary:
   11111100011110100111110000001100
j, int: 198850956, binary:
   00001011110110100011100110001100
i & j, int: 58720644, binary:
   00000011100000000000000110000100
i | j, int: 199212028, binary:
   00001011110111111011101111111100
i ^ j, int: 140491384, binary:
   00001000010111111011101001111000
i << 5, int: 1890614912, binary:
   01110000101100000111111010000000
i >> 5, int: 1846303, binary:
   00000000000111000010110000011111
(~i) >> 5, int: -1846304, binary:
   11111111111000111101001111100000
i >>> 5, int: 1846303, binary:
   00000000000111000010110000011111
(~i) >>> 5, int: 132371424, binary:
   00000111111000111101001111100000
```

數字的二進位表現法是以「帶正負號的二補數（*signed two's complement*）」爲之。

if-else 三元運算子（Ternary if-else operator）

這個運算子比較不尋常，因爲它有三個運算元。由於它會導出一個值，所以它是個如假包換的運算子，和一般的 if-else 述句（下一節介紹）不一樣。其算式格式如下：

```
boolean-exp ? value0 : value1
```

如果上述的 *boolean-exp* 評估爲 **true**，接下來便評估 *value0* 的值，而其評估結果便成爲這個運算子的結果。如果 *boolean-exp* 評估爲 **false**，接下

來便評估 *value1* 的值，而其評估結果便成爲這個運算子的結果。

當然，你也可以使用一般的 **if-else** 述句（稍後提及），但三元運算子更顯精練。雖然 C（三元運算子的濫觴）向來以作爲一個精練的語言自豪，而三元運算子也在某種程度上因效率而被採用，但是你仍然應該在使用時有所警惕 – 是的，它很容易形成不易閱讀的程式碼。

這種所謂條件運算子（conditional operator）的使用目的，也許是爲了其副作用，也許是爲了其運算結果值。一般而言你要的是它的運算結果值，這也正是這個運算子異於 **if-else** 之處。以下便是一例：

```
static int ternary(int i) {
  return i < 10 ? i * 100 : i * 10;
}
```

看得出來，上述程式碼如果不使用三元運算子，不會如此簡潔：

```
static int alternative(int i) {
  if (i < 10)
    return i * 100;
  else
    return i * 10;
}
```

第二種寫法比較容易理解，也不用輸入太多內容。所以，請確定自己在選擇三元運算子時，事先經過周詳考慮。

逗號運算子（comma operator）

C 和 C++語言中的逗號，不僅做爲函式引數列的分隔字元，也做爲「循序評估動作」中的運算子。在 Java 語言中，唯一可以放置逗號運算子的地方，就是 **for** 迴圈（稍後介紹）。

使用於 String 身上的 operator+

在 Java 中，有個運算子提供了很特別的用法：之前你見過的 + 運算子能夠用於字串連接。雖然這種用法不符合傳統，看起來卻頗爲自然。在 C++中，這樣的功能似乎是個不錯的想法，所以「運算子多載化（*operator overloading*）」功能被加至 C++中，允許 C++程式員爲幾乎任何運算子賦予新的意義。不幸的是，運算子多載功能（以及 C++的其他規定），對程式員而言成了一項繁重的負擔。雖然在 Java 中實作「運算子多載化」比起 C++ 來說應該能夠更簡單，但這個功能依舊被認爲過度複雜，所以 Java 不允許程式員實作他們自有的多載化運算子。

String + 的運用有一些很有趣的行爲。如果某個算式以 **String** 爲首，那麼接續的所有運算元也都必須是 **String**s（別忘了，編譯器會將雙引號括住的字元序列自動轉化爲 **String**）：

```
int x = 0, y = 1, z = 2;
String sString = "x, y, z ";
System.out.println(sString + x + y + z);
```

在這裡，Java 編譯器會將 **x**、**y**、**z** 轉化爲它們各自的 **String** 表示式，而不是先將它們加在一起。如果你這麼寫：

```
System.out.println(x + sString);
```

Java 會將 **x** 轉換爲 **String**。

使用運算子時的常犯錯誤

使用運算子時，一個常犯的錯誤便是，雖然你對運算式的評估方式有點不確定，卻不願意使用小括號來幫助自己。這句話在 Java 之中仍然成立。

在 C 和 C++中一個極常見的錯誤如下：

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

程式員想做的其實是相等測試（==）而非賦值動作，卻打錯了字。在 C 和 C++ 中，如果 y 值非零，那麼此一賦值動作的結果肯定為 **true**，你因此陷入一個無窮迴圈。在 Java 中，此一算式的結果並不是 **boolean**，但編譯器預期此處應該是個 **boolean**，並且不希望由 **int** 轉換過來。因此編譯器會給你適當的錯誤訊息，在你嘗試執行程式之前捕捉到這個問題。所以這樣的陷阱不會出現在 Java 中。如果 **x** 和 **y** 都是 **boolean**，你不會獲得編譯錯誤訊息，因為 **x = y** 是合法算式，但這卻不是你所想像的情況。

C 和 C++ 還有一個類似問題：容易將位元運算子 AND 和 OR 誤為對應的邏輯運算子。位元運算子 AND 和 OR 採用的符號是單一字元（& 或 |），邏輯運算子 AND 和 OR 則採用雙字元符號（&& 或 ||）。這種情況很像 = 和 ==，很容易因為疏忽而只鍵入一個符號。Java 編譯器也會杜絕此一問題，它不讓你的漫不經心破壞編程大計。

轉型運算子（Casting operators）

cast 這個字源於 "casting into a mold"（鑄入一個模子內）。Java 能夠在適當時機自動將資料從某個型別改變為另一個型別。舉例來說，如果你將整數值指派給浮點變數，編譯器便會自動將 **int** 轉換為 **float**。不過，轉型（*casting*）讓你可以更明確地進行這類型別轉換，或者讓你在原本不會自然發生的場合，強迫它發生。

欲執行轉型動作，請將標的型別（包括所有飾詞）置於任意數值的左方括號內。以下便是一例：

```
void casts() {
  int i = 200;
  long l = (long)i;
  long l2 = (long)200;
}
```

如你所見，將數值轉型，就和將變數轉型一樣，是可能的。上述例子同時展現出兩種轉型方式。這樣的轉型其實是多餘的，因為編譯器會在必要時候自動將 **int** 晉升（promote）為 **long**。不過，為了表明某種觀點，或是

爲了讓程式碼更明瞭易讀，這類非必要的轉型是允許的。至於其他情況，有可能必須先完成轉型動作才可以順利通過編譯。

在 C 和 C++ 中，轉型可能引發其他麻煩事兒。Java 中的轉型是安全的，只有當執行所謂窄化轉換（*narrowing conversion*）（也就是說當你將某個存有較多資訊的資料型別轉換爲無法儲存那麼多資訊的另一個型別時）才有風險 — 彼時你得冒著資訊遺失的風險。這種情形下編譯器會強迫你進行轉型。實際上它會說：『這麼做可能是危險的。如果你希望我放手一搏，你得明確給我指示』。進行寬化轉換（*widening conversion*）時就無需明確指示，因爲新型別能夠容納來自舊型別的資訊，不會遺失任何資訊。

Java 允許你將任意基本型別轉型爲另一個任意基本型別。然而 **boolean** 例外，它完全不接受任何轉型動作。class 型別不允許轉型。想要將某個 class 型別轉換爲另一個 class 型別，得有特殊方法才辦得到。**String** 是個特例。本書稍後你還會看到，同一 classes 族系的物件之間可以轉型。是的，**Oak**（橡樹）可被轉爲 **Tree**（樹木），反之亦然。但我們無法將它轉爲族系以外的型別，例如 **Rock**（石頭）。

字面常數（Literals）

一般而言，當你將某個常數値置於程式中，編譯器很清楚知道要將它製成什麼型別。但有時候難免模稜兩可。一旦發生這種情形，你得提供某些額外資訊：透過「爲常數値搭配某些字元」的形式，引導編譯器做出正確的判斷。以下程式碼展示這種情況下的搭配字元：

```
//: c03:Literals.java

class Literals {
  char c = 0xffff; // max char hex value
  byte b = 0x7f; // max byte hex value
  short s = 0x7fff; // max short hex value
  int i1 = 0x2f; // Hexadecimal (lowercase)
  int i2 = 0X2F; // Hexadecimal (uppercase)
  int i3 = 0177; // Octal (leading zero)
  // Hex and Oct also work with long.
```

```
  long n1 = 200L; // long suffix
  long n2 = 200l; // long suffix
  long n3 = 200;
  //! long l6(200); // not allowed
  float f1 = 1;
  float f2 = 1F; // float suffix
  float f3 = 1f; // float suffix
  float f4 = 1e-45f; // 10 to the power
  float f5 = 1e+9f; // float suffix
  double d1 = 1d; // double suffix
  double d2 = 1D; // double suffix
  double d3 = 47e47d; // 10 to the power
} ///:~
```

十六進制（以 16 爲基底）能夠應用於所有整數資料型別，其表示法係以 **0x** 或 **0X** 爲首，後接 0-9 或 a-f（大小寫皆可）。如果你嘗試將某變數的初值設成比該變數所能儲存的最大值還大的話（不論是以八進制、十進制或十六進制），編譯器會給你錯誤訊息。請注意上述程式碼中 **char**、**byte**、**short** 的最大可能十六進制值。如果超過這些值，編譯器會自動將該值視爲 **int**，並告訴你此一賦值動作需要窄化轉換（narrowing conversion）。這時候你就知道自己越界了。

八進制（以 8 爲基底）係以 **0** 爲首，每一位數皆落在 0-7 中。C、C++、Java 都沒有提供二進制的數字常數表示法。

常數值之後添增的字元係用來表明數值的型別。大寫或小寫的 **L** 意指 **long**，大寫或小寫的 **F** 意指 **float**，大寫或小寫的 **D** 意指 **double**。

指數（exponents）採用的是一種向來讓我感到驚恐的表示法：**1.39e-47f**。在科學和工程領域中，'e' 所代表的是自然對數的基底，近似於 2.718（Java 裡頭有個更精確的 **double** 值是 **Math.E**），它被用於像 1.39 x e^{-47}這樣的指數表示式，意指 1.39 x 2.718^{-47}。但是 FORTRAN 發明之際，那些人決定讓 e 代表「10 的次方」。這是個相當怪異的決定，因爲 FORTRAN 乃是被設計用於科學和工程領域，而任何人都可能認爲，

FORTRAN 的設計者在引入這樣的歧義[1] 時，必然經過審慎的思考。無論如何，這個慣例被 C、C++、Java 採用了。所以如果你習慣將 **e** 思考爲自然對數基底，那麼當你在 Java 中看到諸如 **1.39e-47f** 這樣的表示式時，你得在心中默默提醒自己：它指的其實是 1.39×10^{-47}。

請注意，如果編譯器能夠找出適當的型別，你就不需要在數值之後附加字元，那將是一種非必要的補充。以下寫法並不會造成含糊不清的狀況：

```
long n3 = 200;
```

所以 200 之後的 **L** 便顯多餘。但如果寫成這樣：

```
float f4 = 1e-47f; // 10 to the power
```

編譯器會順理成章地將指數視爲 **doubles**。所以，由於少了附加字元 **f**，編譯器會給你錯誤訊息，讓你知道，你得透過轉型動作，將 **double** 轉換爲 **float**。

晉升（Promotion）

你會發現，當你在比 **int** 更小的基本型別（亦即 **char**、**byte**、**short**）上進行任何數學運算或位元運算時，運算之前其值會先被晉升爲 **int**，最後所得結果也會是 **int** 型別。因此如果你想要將結果指派給較小型別，就得進行轉型動作。而且由於指派的目的地是較小型別，有可能遺失資訊。一般來說，算式內出現的最大資料型別，是決定該算式運算結果的容量大小的

[1] John Kirkham 是這麼寫的：『我於 1962 年首次在 IBM 1620 電腦上以 FORTRAN II 進行計算。1760-1970 年代，FORTAN 是個全面使用大寫字母的語言。這或許是因爲早期許多輸入設備都是使用 5 bit Baudot code 的舊式電報裝置，此類裝置沒有小寫功能。指數表示式中的 'E' 也絕對是大寫，從未和必爲小寫的自然對數基底 'e' 混淆過。'E' 僅僅只是代表指數（exponential），代表所使用的數値系統的基底 – 通常是 10。當時八進位亦被程式員廣泛使用，雖然我未曾見過，但如果當時我曾經見過指數表示式中以八進位數字來表達的話，我會考慮讓它以 8 爲基底。我首次看到使用小寫 'e' 的指數表示式，是在 1970 年代末期，我也認爲這會產生混淆。問題發生在「小寫字母逐漸進入 FORTRAN」之際，而非發生在 FORTRAN 一開始發展之時。如果你真想以自然對數做爲基底，我們另外提供了函式，不過它們也全都是大寫名稱。』

依據之一；如果你讓 **float** 和 **double** 相乘，結果便是 **double**；如果你讓 **int** 和 **long** 相加，結果便是 **long**。

Java 沒有 "sizeof" 運算子

C 和 C++ 的 **sizeof()** 運算子滿足了一個特定需求：它讓你知道，編譯器究竟爲某一筆資料配置了多少個 bytes。**sizeof()** 在 C 和 C++ 中的存在必要性，最令人信服的理由便是爲了可攜性。不同的資料型別在不同的機器上可能會有著不同的大小，所以執行「和容量大小有高度相關」的運算時，程式員必須知道這些型別的容量究竟有多大。例如某部電腦可能以 32 bits 來儲存整數值，另一部電腦卻可能以 16 bits 來儲存整數值；程式在第一部機器上能夠以整數儲存較大的值。就如你所能想像的，可攜性對 C 和 C++ 程式員來說，是個棘手的問題。

Java 不需要爲了這個原因而提供 **sizeof()** 運算子。因爲在所有機器上，每一種資料型別都有著相同的容量大小。在這個層次上，你完全不需要思考可攜性的問題 – 它已被設計於語言之中。

再探優先序（precedence）

在我的某個研討班上，當我抱怨運算子優先序過於複雜而難以記憶之後，有位學生向我推薦一種助憶法，這個口訣像是一句評語：Ulcer Addicts Really Like C A lot（胃潰瘍患者是 C 程式員的寫照）。

助憶口訣	運算子類型	運算子
Ulcer	Unary	`+ - ++--`
Addicts	Arithmetic (以及 shift)	`* / % + - << >>`
Really	Relational	`> < >= <= == !=`
Like	Logical (以及 bitwise)	`&& || & | ^`
C	Conditional (三元)	`A > B ? X : Y`
A Lot	Assignment	`=` (以及複合指派動作如 `*=`)

當然，讓「位移運算子」和「位元運算子」散落於表格四處，並不是一種完美的助憶法，但對於非位元運算來說，沒有問題。

運算子綜合說明

以下範例說明哪些基本型別能被施行哪些特定運算子。基本上這是一個一再重覆的程式，只不過每次運用不同的基本型別。此程式能夠順利通過編譯，沒有任何錯誤訊息，因爲，可能導致錯誤的每一行程式都被我以 //! 化爲註解了。

```
//: c03:AllOps.java
// Tests all the operators on all the
// primitive data types to show which
// ones are accepted by the Java compiler.

class AllOps {
  // To accept the results of a boolean test:
  void f(boolean b) {}
  void boolTest(boolean x, boolean y) {
    // Arithmetic operators:
    //! x = x * y;
    //! x = x / y;
    //! x = x % y;
    //! x = x + y;
    //! x = x - y;
    //! x++;
    //! x--;
    //! x = +y;
    //! x = -y;
    // Relational and logical:
    //! 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;
    // Bitwise operators:
    //! x = ~y;
    x = x & y;
    x = x | y;
```

```
    x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Compound assignment:
    //! 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) {
    // Arithmetic operators:
    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;
    // Relational and logical:
    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);
    // Bitwise operators:
    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);
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! boolean b = (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) {
    // Arithmetic operators:
    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;
    // Relational and logical:
    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);
    // Bitwise operators:
    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);
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! boolean b = (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) {
    // Arithmetic operators:
    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;
    // Relational and logical:
    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);
    // Bitwise operators:
    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);
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
```

```
    x ^= y;
    x |= y;
    // Casting:
    //! boolean b = (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) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    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);
    // Bitwise operators:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    x = x >>> 1;
    // Compound assignment:
```

```
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! boolean b = (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) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    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);
```

```
    // Bitwise operators:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! boolean b = (boolean)x;
    char c = (char)x;
    byte B = (byte)x;
    short s = (short)x;
    int i = (int)x;
    float f = (float)x;
    double d = (double)x;
  }
  void floatTest(float x, float y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
```

```
    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);
    // Bitwise operators:
    //! x = ~y;
    //! x = x & y;
    //! x = x | y;
    //! x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    //! x &= y;
    //! x ^= y;
    //! x |= y;
    // Casting:
    //! boolean b = (boolean)x;
    char c = (char)x;
    byte B = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    double d = (double)x;
  }
  void doubleTest(double x, double y) {
    // Arithmetic operators:
    x = x * y;
```

```
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    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);
    // Bitwise operators:
    //! x = ~y;
    //! x = x & y;
    //! x = x | y;
    //! x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    //! x &= y;
    //! x ^= y;
    //! x |= y;
    // Casting:
    //! boolean b = (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;
  }
} ///:~
```

請注意，能夠在 **boolean** 身上進行的動作極爲有限。你可以賦予其值爲 **true** 或 **false**，也可以檢驗其值是否爲真，但你無法將兩個 **boolean**s 相加，或在它們身上執行其他形式的運算。

在 **char**、**byte**、**short** 身上，你可以看到施行算術運算子時所發生的晉升（promotion）效應。施行於這些型別身上的任何算術運算，皆回傳 **int**，因而必須明確將它轉回原先型別。操作 **int** 時，弗需動用轉型，因爲每個運算元都已經是 **int**。但千萬別因此鬆懈，進而以爲每件事情都安全牢靠。如果你將兩個夠大的 **int**s 相乘，結果便會溢位（overflow）。以下程式碼展現了這一點：

```
//: c03:Overflow.java
// Surprise! Java lets you overflow.

public class Overflow {
  public static void main(String[] args) {
    int big = 0x7fffffff; // max int value
    prt("big = " + big);
    int bigger = big * 4;
    prt("bigger = " + bigger);
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

輸出結果是：

```
big = 2147483647
bigger = -4
```

而且你不會在編譯期得到任何錯誤訊息或警告訊息，也不會在執行期得到任何異常（exception）。Java 是很好沒錯，但沒那麼好。

char、**byte**、**short** 的複合賦值運算（compound assignments）不需要轉型，雖然，它們都進行了晉升動作並與直接（非複合）運算有相同的結果。從另一個角度看，少了轉型動作，可以簡化程式碼。

你也看到了，除了 **boolean** 之外，所有基本型別都可以被轉型爲其它任意基本型別。此外，你必須清楚轉型至較小型別時所發生的窄化效應，否則資訊也許會在轉型過程中被你不知不覺地遺失掉。

流程控制

Java 採納 C 語言的所有流程控制述句。所以如果你曾經有過 C 或 C++ 編程經驗，此刻你所見到的幾乎都是你已經熟悉的語法。許多程序式（procedural）語言都具備某些類型的控制述句，它們在許多語言之間常有重疊。Java 的相應關鍵字包括了 **if-else**、**while**、**do-while**、**for**、**switch-case**。Java 並未提供 **goto** – 一個被過度中傷的東西（在解決某些類型的問題上，它仍然是最權宜的方式）。你還是可以在 Java 程式中進行類似 **goto** 的跳躍行爲，但比起典型的 **goto** 來說，受限很多。

true 和 false

所有條件述句都使用某個條件算式的運算結果（真僞值）來決定程式的執行路徑。條件算式就像 **A == B** 這樣。這個例子利用條件運算子 == 來檢驗 **A** 值是否等於 **B** 值，並回傳 **true** 或 **false**。本章稍早出現的所有關係運算子，都可被用來產生條件述句。請注意，雖然 C 和 C++ 允許使用數字做爲 **boolean**（它們視非零值爲真，零值爲僞），Java 卻不允許這麼做。如果你想在 **boolean** 測試中使用 non-**boolean** 值，例如 **if(a)**，你得先以條件算式將它轉換爲 **boolean** 值，例如 **if(a != 0)**。

if-else

if-else 述句或許是控制程式流程的眾多方法中最基本的一個。**else** 子句可有可無，所以你可以採取兩種形式來使用 **if**：

```
if(Boolean-expression)
  statement
```

或是：

```
if(Boolean-expression)
  statement
else
  statement
```

其中的條件句必須得出 **boolean** 結果。*statement* 意指單述句（以分號做結尾）或複合述句（以成對大括號括住的一組單述句）。任何時候當我使用 *statement* 這個字，我的意思便是指單述句或複合述句。

以下的 **test()** 用來示範 **if-else** 的運用。它能夠告訴你你所猜的數字究竟大於、小於、或等於謎底：

```
//: c03:IfElse.java
public class IfElse {
  static int test(int testval, int target) {
    int result = 0;
    if(testval > target)
      result = +1;
    else if(testval < target)
      result = -1;
    else
      result = 0; // Match
    return result;
  }
  public static void main(String[] args) {
    System.out.println(test(10, 5));
    System.out.println(test(5, 10));
    System.out.println(test(5, 5));
  }
} ///:~
```

習慣上我們會將流程控制式中的 *statement* 加以縮排，這麼一來讀者更能夠輕易判斷其啓始處和終止處。

return

關鍵字 **return** 有兩個用途：指明某個函式即將傳回之值（如果回傳型別不爲 **void** 的話），並令該值立即被傳回。我們可將上例中的 **test()** 重新改寫以發揮此一優點：

```
//: c03:IfElse2.java
public class IfElse2 {
  static int test(int testval, int target) {
    int result = 0;
    if(testval > target)
      return +1;
    else if(testval < target)
      return -1;
    else
      return 0; // Match
  }
  public static void main(String[] args) {
    System.out.println(test(10, 5));
    System.out.println(test(5, 10));
    System.out.println(test(5, 5));
  }
} ///:~
```

現在我們不需要 **else** 了，因爲 **test()** 在執行 **return** 之後不再繼續執行。

迭代（iteration）

while、**do-while**、**for** 三組關鍵字用來控制迴圈（loop），它們有時被歸類爲迭代述句（*iteration statements*）。*statement* 會反覆執行，直到控制用的 *Boolean-expression* 被評估爲 **false** 才停止。**while** 迴圈形式是：

```
while(Boolean-expression)
  statement
```

Boolean-expression 在迴圈開始時會被評估一次，並且在每次執行完 *statement* 後，再評估一次。

以下是個簡單範例，持續產生亂數，直到特定條件滿足爲止：

```
//: c03:WhileTest.java
// Demonstrates the while loop.

public class WhileTest {
  public static void main(String[] args) {
    double r = 0;
    while(r < 0.99d) {
      r = Math.random();
      System.out.println(r);
    }
  }
} ///:~
```

這個例子使用 **Math** 程式庫中的 **static random()**，產生介於 0 與 1 之間的 **double** 值（包含 0 但不包含 1）。**while** 的條件算式所陳述的是「持續執行迴圈，直到此數字爲 0.99 或更大」。每當你執行此一程式，你都會得到一連串個數不盡相同的數字。

do-while

do-while 的形式是：

```
do
  statement
while(Boolean-expression);
```

while 和 **do-while** 之間的唯一差別在於：**do-while** 中的述句至少執行一次，即使算式一開始就被評估爲 **false**。但是在 **while** 中，如果條件句一開始就是 **false**，迴圈內的述句完全不會被執行。實務應用上 **do-while** 遠比 **while** 罕見。

for

for 迴圈在首次迭代前，會先進行初始化動作。然後進行條件測試，並在每執行完一次迭代，就執行某種形式的「步進（stepping）」動作。**for** 迴圈形式如下：

```
for(initialization; Boolean-expression; step)
  statement
```

其中的 *initialization*、*Boolean-expression* 或 *step* 算式皆可爲空。每次迭代前會檢驗算式值，並在該算式評估爲 **false** 後立刻執行緊接於 **for** 述句之後的下一行程式。每次迭代結束，*step* 便會被執行。

for 迴圈通常被用於「計數」工作：

```
//: c03:ListCharacters.java
// Demonstrates "for" loop by listing
// all the ASCII characters.

public class ListCharacters {
  public static void main(String[] args) {
  for( char c = 0; c < 128; c++)
    if (c != 26 )  // ANSI Clear screen
      System.out.println(
        "value: " + (int)c +
        " character: " + c);
  }
} ///:~
```

請注意，變數 **c** 定義於它被使用之處，也就是在 **for** 迴圈的控制算式內，而不在成對大括號所標記的區段起始處。**c** 的可見範圍落在由 **for** 所控制的算式中。

諸如 C 之類的傳統程序式（procedural）語言，要求所有變數都必須被定義於區段起始處。這麼一來編譯器建立區段時，才能夠爲這些變數配置空間。但是在 Java 和 C++ 中，你可以將變數的宣告式置於整個區段的任意位置，並在需要用到它們時才加以定義。如此一來編程風格更趨自然，程式碼更易閱讀。

你可以在 **for** 述句中定義多個變數，但它們的型別必須一致：

```
for(int i = 0, j = 1;
    i < 10 && j != 11;
    i++, j++)
  /* body of for loop */;
```

for 述句中的 **int** 定義式，同時涵蓋了 **i** 和 **j**。只有 **for** 迴圈才擁有「在控制算式中定義變數」的能力。你無法將這種寫法套用於其它類型的選擇述句或迭代述句身上。

逗號運算子（comma operator）

本章稍早，我曾提過逗號運算子（不是作爲分隔多個變數定義或多個函式引數的那個所謂逗號分隔器）。這個運算子在 Java 中僅有一種用法：用於 **for** 迴圈的控制算式。是的，在 **for** 迴圈的 *initialization* 和 *step* 兩部份中，都可以存在多個由逗號分隔的述句，而且這些述句會被依序評估。上一小段程式碼便使用了這項能力。以下是另一個例子：

```
//: c03: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);
    }
  }
} ///:~
```

輸出結果如下：

```
i= 1 j= 11
i= 2 j= 4
i= 3 j= 6
i= 4 j= 8
```

你可以看到，在 *initialization* 和 *step* 兩部份中的述句被依序評估。而且，*initialization* 所含的「相同型別」的變數定義，個數不限。

break 和 continue

在迭代述句的主體內，你隨時可以使用 **break** 和 **continue** 來控制迴圈流程。**break** 會跳出迴圈，不再執行剩餘部份。**continue** 會停止當次迭代，回到迴圈起始處，開始下一個迭代過程。

下面這個程式示範在 **for** 和 **while** 迴圈中使用 **break** 和 **continue** 的方式：

```
//: c03:BreakAndContinue.java
// Demonstrates break and continue keywords.

public class BreakAndContinue {
  public static void main(String[] args) {
    for(int i = 0; i < 100; i++) {
      if(i == 74) break; // Out of for loop
      if(i % 9 != 0) continue; // Next iteration
      System.out.println(i);
    }
    int i = 0;
    // An "infinite loop":
    while(true) {
      i++;
      int j = i * 27;
      if(j == 1269) break; // Out of loop
      if(i % 10 != 0) continue; // Top of loop
      System.out.println(i);
    }
  }
} ///:~
```

在 **for** 迴圈中，**i** 的值永遠不會是 100，因爲 **break** 述句會在 **i** 值爲 74 時中斷迴圈的進行。通常，只有在你不知道終止條件究竟何時發生時，才應該使用 **break**。每當 **i** 不能被 10 整除，**continue** 述句便會將執行點移至迴圈最前端（因而將 **i** 累加 1）；如果整除，該值便會被印出。

第二部份示範了理論上持續不止的「無窮迴圈（infinite loop）」。不過，迴圈中有個 **break** 述句，可以中斷迴圈。此外你也會看到 **continue** 將迴圈的執行點移回頂端，不再完成剩餘部份（因此在第二個迴圈中，只有當 **i** 值被 10 整除時，列印動作才會發生）。輸出結果如下：

```
0
9
18
27
36
45
54
63
72
10
20
30
40
```

其中之所以會印出 0 值，乃是因爲 0 % 9 獲得 0。

無窮迴圈的第二種形式是 **for(;;)**。編譯器將 **while(true)** 和 **for(;;)** 視爲相同。所以，究竟選用哪一個，是編程品味的問題。

惡名昭彰的 “goto”

自從第一個程式語言以來，關鍵字 **goto** 便已存在。的確，組合語言中的 **goto** 是程式流程控制的濫觴：「如果條件 A 成立，就跳到這兒來，否則就跳到那兒」。如果你讀過隨便哪個編譯器最終產生出來的組合語言碼，你會發現，在程式流程控制之處，含有許多跳躍動作（jumps）。不過我們現在所談的 **goto**，卻是原始碼層次上的跳躍，而這正是其惡名昭彰的源頭。如果程式總是從某一點跳躍至另一點，有什麼方法可以整頓程式碼，使流程控制不會變得如此變化多端？由於 Edsger Dijkstra 發表了一篇著名論文《*Goto considered harmful*》，**goto** 從此陷入萬劫不復的境地。而由於對 **goto** 的攻訐成爲一種普世運動，眾人於是倡言將此關鍵字徹底逐出語言門牆。

一般來說，在這種情況下，中庸之道永遠是最好的一條路。問題不在於 **goto** 的使用，而在於 **goto** 的過度使用 － 極少數情況下，**goto** 其實是流程控制的最佳方法。

雖然 **goto** 是 Java 保留字，但這個語言並沒有使用它；是的，Java 裡頭沒有用到 **goto**。不過 Java 卻有一些看起來有點像跳躍動作（jump）的功能，這個功能和關鍵字 **break** 以及關鍵字 **continue** 結合在一起。它們其實並不是跳躍，而是一種中斷迭代述句的方式。它們之所以被拿來和 **goto** 相提並論，因爲它們使用了相同的機制：label（標記）。

所謂 label，是個後面緊接冒號的識別字，就像這樣：

```
label1:
```

在 Java 中，唯一一個「置放 label，而能夠產生效益」的地點，就是恰恰放在迭代述句之前。在 label 與迭代內容之間安插任何述句，不會帶來什麼好處。將 label 置於迭代述句之前，完全是爲了對付那種「巢狀進入另一個迭代或 switch」的情況。因爲，關鍵字 **break** 和 **continue** 一般而言只會中斷當次迴圈，如果搭配 label 使用，它們會中斷所有進行中的巢狀迴圈，直達 label 所在處：

```
label1:
outer-iteration {
  inner-iteration {
    //…
    break; // 1
    //…
    continue;  // 2
    //…
    continue label1; // 3
    //…
    break label1;  // 4
  }
}
```

請看上例狀況 1，**break** 會中斷內層迭代，回到外層迭代。狀況 2 的 **continue** 會將執行點移至內層迭代的起始處。狀況 3 的 **continue label1** 會同時中斷內層和外層迭代，直接回到 **label1**，此時迭代動作繼續進行，但卻從外層迭代（而非內層迭代）重新開始。狀況 4 的 **break lable1** 也是跳脫一切約束，移至 **label1**，但是不會再度進入迭代 — 它同時中斷了內外層迭代。

以下是 label 搭配 **for** 迴圈的使用實例：

```
//: c03:LabeledFor.java
// Java's "labeled for" loop.

public class LabeledFor {
  public static void main(String[] args) {
    int i = 0;
    outer: // Can't have statements here
    for(; true ;) { // infinite loop
      inner: // Can't have statements here
      for(; i < 10; i++) {
        prt("i = " + i);
        if(i == 2) {
          prt("continue");
          continue;
        }
        if(i == 3) {
          prt("break");
          i++; // Otherwise i never
               // gets incremented.
          break;
        }
        if(i == 7) {
          prt("continue outer");
          i++; // Otherwise i never
               // gets incremented.
          continue outer;
        }
        if(i == 8) {
          prt("break outer");
          break outer;
        }
        for(int k = 0; k < 5; k++) {
          if(k == 3) {
            prt("continue inner");
            continue inner;
          }
        }
      }
    }
```

```
    // Can't break or continue
    // to labels here
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

這個例子用到了定義於其他例子中的 **prt()**。

請注意，本例之中，**break** 會跳出 **for** 迴圈；由於沒有經歷一次完整的迭代，所以迴圈的累進算式（*increment expression*）不會發生。正因爲 **break** 會略過累進算式，爲了彌補，我們在 **i == 3** 的情況下直接將 **i** 加 1。當 **i == 7** 時，**continue outer** 述句也會跳回迴圈頂端，而且也會略去累進動作，所以我們也直接在該情況下將 **i** 累加 1，以利程式進行。

本例輸出如下：

```
i = 0
continue inner
i = 1
continue inner
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
```

如果缺少 **break outer** 述句，我們就沒有任何辦法在內層迴圈中直接跳離外層迴圈，因爲 **break** 僅能中斷最內層迴圈（**continue** 亦然）。

當然，如果你想在中斷迴圈時一併離開函式，只要使用 **return** 即可。

下面這個例子，示範如何在 **while** 迴圈中將 **break** 述句和 **continue** 述句搭配 label 來使用：

```
//: c03:LabeledWhile.java
// Java's "labeled while" loop.

public class LabeledWhile {
  public static void main(String[] args) {
    int i = 0;
    outer:
    while(true) {
      prt("Outer while loop");
      while(true) {
        i++;
        prt("i = " + i);
        if(i == 1) {
          prt("continue");
          continue;
        }
        if(i == 3) {
          prt("continue outer");
          continue outer;
        }
        if(i == 5) {
          prt("break");
          break;
        }
        if(i == 7) {
          prt("break outer");
          break outer;
        }
      }
    }
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

對 **while** 而言，規則依舊成立：

1. 一般的 **continue** 會回到最內層迴圈頂端，繼續執行。
2. *labeld* **continue** 會跳躍至 label 所在處，然後恰在 label 之後重新進入迴圈。
3. 一般的 **break** 會跳離迴圈。
4. *labeld* **break** 會跳離 label 所描述的迴圈。

上個例子的輸出結果更能清楚說明以上所言：

```
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
```

請務必記住，在 Java 裡頭使用 labels，唯一的理由是：在巢狀迴圈中想要令 **break** 或 **continue** 越過一個以上的巢狀層級（nested level）。

Dijkstra 的論文《*goto considered harmful*》，反對的其實是 labels 而非 **goto**。他觀察到，程式臭蟲的數目似乎隨著程式中的 labels 個數而成長。labels 和 **goto** 使程式難以被靜態分析，因爲它們會將循環（cycles）引入程式執行圖（execution graph）中。請記住，Java 的 labels 不會帶來這種問題，因爲它們能夠擺放的位置有限，此外它們也不能隨意被用來改變執行流程。這是「藉由侷限某個述句的威力，讓語言的某種性質更爲有用」的一個有趣例子。

switch

switch 有時候亦被歸類爲「選擇述句（*selection statement*）」。**switch** 述句會根據某個整數算式的值，在眾多程式碼片段中挑出一段來執行。形式如下：

```
switch(integral-selector) {
  case integral-value1 : statement; break;
  case integral-value2 : statement; break;
  case integral-value3 : statement; break;
  case integral-value4 : statement; break;
  case integral-value5 : statement; break;
          // ...
  default: statement;
}
```

其中 *integral-selector* 是個能得出整數值的算式。**switch** 會拿 *integral-selector* 和每個 *integral-value* 逐一比較，如果找到吻合者，就執行相應的 *statement*（不論是單述句或複合述句）。如果找不到吻合者，就執行 *default statement*。

你應該注意到了，上述定義中，每個 **case** 皆以 **break** 做爲結束，這會使程式執行點跳至 **switch** 本體的最末端。此爲建構 **switch** 述句最常見的形式，然而 **break** 的存在並非絕對必要。如果少了 **break**，便會執行其後接續的 **case** 述句，直到遇上 **break** 爲止。雖然你通常不會希望這種行爲發生，但對有經驗的程式員來說，這種性質相當有用。請注意，**default** 之後的最後一個述句並未加上 **break**，因爲即使沒有放上 **break**，程式也是恰好執行到應該去的地點，所以沒有差別。如果你考慮編程風格的問題，可以將 **break** 置於 **default** 述句之末，那也無妨。

在多向選擇（multi-way selection，也就是從多個不同的執行路徑中挑選一個）的實作手法上，**switch** 述句很是乾淨俐落，但是你得有個「能核定出整數值（例如 **int** 或 **char**）」的選擇器（*selector*）才行。如果你想以字串或浮點數做爲選擇器，在 **switch** 述句中是行不通的。面對非整數型別，你非得使用一連串 **if** 述句不可。

下面這個例子會隨機產生字母，並判斷該字母是母音或子音：

```
//: c03:VowelsAndConsonants.java
// Demonstrates the switch statement.

public class VowelsAndConsonants {
  public static void main(String[] args) {
    for(int i = 0; i < 100; i++) {
      char c = (char)(Math.random() * 26 + 'a');
      System.out.print(c + ": ");
      switch(c) {
      case 'a':
      case 'e':
      case 'i':
      case 'o':
      case 'u':
                System.out.println("vowel");
                break;
      case 'y':
      case 'w':
                System.out.println(
                  "Sometimes a vowel");
                break;
      default:
                System.out.println("consonant");
      }
    }
  }
} ///:~
```

由於 **Math.random()** 會產生介於 0 和 1 之間的值，所以你只要將「所產生的隨機值」乘以「欲得的範圍上限」（對字母系統而言爲 26），再加上一個偏移量，即可獲得隨機字母。

雖然此例是在字元身上進行選擇，但 **switch** 述句所用的其實是該字元的整數值。**case** 述句中以單引號括起來的字元，亦以其整數值進行比較。

請注意，多個 **case**s 能夠一個個堆疊起來，造成「只要吻合其中任何一個條件，便都執行相同的程式碼」的效果。運用此技巧時你應該知道，將 **break** 述句置於特定 **case** 之後是有必要的，否則程式流程便會繼續往下執行，處理下一個 **case**。

計算細節

以下述句值得細細端詳：

```
char c = (char)(Math.random() * 26 + 'a');
```

Main.random() 會得出一個 **double** 值，所以 26 會被轉換成 **double** 以利乘法運算，而乘積亦是個 **double**。這同時也意謂為了完成加法，**'a'** 必須被轉換為 **double**。最後所獲得的 **double** 再被轉為 **char**。

將 **double** 轉為 **char** 的過程中會進行哪些動作呢？如果將 29.7 轉型至 **char**，得到的是 30 或是 29？此一問題的答案可由下例瞧出端倪：

```
//: c03:CastingNumbers.java
// What happens when you cast a float
// or double to an integral value?

public class CastingNumbers {
  public static void main(String[] args) {
    double
      above = 0.7,
      below = 0.4;
    System.out.println("above: " + above);
    System.out.println("below: " + below);
    System.out.println(
      "(int)above: " + (int)above);
    System.out.println(
      "(int)below: " + (int)below);
    System.out.println(
      "(char)('a' + above): " +
      (char)('a' + above));
    System.out.println(
      "(char)('a' + below): " +
```

```
      (char)('a' + below));
  }
} ///:~
```

輸出結果是：

```
above: 0.7
below: 0.4
(int)above: 0
(int)below: 0
(char)('a' + above): a
(char)('a' + below): a
```

所以，先前的答案是：從 **float** 或 **double** 轉爲整數値，總是以完全捨棄小數（而非四捨五入）的方式進行。

第二個問題和 **Math.random()** 有關。它所產生的介於 0 與 1 之間的值，究竟包不包括 '1'？以數學行話來說，它究竟是(0,1)、[0,1]、(0,1]、或是[0,1)？（中括號意指「包含」，小括號意指「不包含」。）寫個測試程式就知道了：

```
//: c03:RandomBounds.java
// Does Math.random() produce 0.0 and 1.0?

public class RandomBounds {
  static void usage() {
    System.out.println("Usage: \n\t" +
      "RandomBounds lower\n\t" +
      "RandomBounds upper");
    System.exit(1);
  }
  public static void main(String[] args) {
    if(args.length != 1) usage();
    if(args[0].equals("lower")) {
      while(Math.random() != 0.0)
        ; // Keep trying
      System.out.println("Produced 0.0!");
    }
    else if(args[0].equals("upper")) {
      while(Math.random() != 1.0)
        ; // Keep trying
```

```
        System.out.println("Produced 1.0!");
      }
      else
        usage();
    }
} ///:~
```

欲執行此程式，只須在命令列（command line）鍵入以下二者之一即可：

```
java RandomBounds lower
```

或

```
java RandomBounds upper
```

無論哪一種情況你都得手動中斷程式，這樣看起來，**Math.random()** 絕對不會產生 0.0 或 1.0 兩個值。但這正是這種實驗可能作弊的地方。如果你能夠想像 0 和 1 之間共有 2^{62} 種不一樣的 **double** 值[2]，以上述實驗方式得到某值，所耗費的時間可能會超過一部電腦甚至一個實驗者的壽命。事實的真象是，**Math.random()** 的輸出包括 0.0，以數學術語來說，其輸出範圍是 [0,1)。

摘要

本章歸納整理了大多數程式語言皆有的基礎功能：計算、運算子優先序、型別轉換、流程選擇和迭代。現在，你已經準備好往前踏出一步，使你自

---

[2] Chuck Allison 是這麼說的：在一個浮點數系統中，可表達之數字的個數是：
2(M-m+1)b^(p-1) + 1，其中 **b** 是基底（通常爲 2），**p** 是精確度（假數（mantissa）中的位數），**M** 是指數的最大值，**m** 是指數的最小值。IEEE 754 的規範是：**M = 1023, m = -1022, p = 53, b = 2**，所以能夠表現的數字總共有這麼多個：
2(1023+1022+1)2^52
= 2((2^10-1) + (2^10-1))2^52
= (2^10-1)2^54
= 2^64 - 2^54
其中半數（指數值落在[-1022,0]範圍內）的值（包括正數和負數）小於 1。所以上述表示式的 1/4，也就是 2^62 - 2^52 + 1（接近 2^62）落在 [0,1)範圍中。請參考我置於 http://www.freshsources.com/1995006a.htm 上的論文。

己更靠近物件導向程式設計的世界。下一章將涵蓋物件（objects）的初始化和清理。下下一章則討論實作隱藏（implementation hiding）的核心觀念。

練習

某些經過挑選的題目，其解答置於《*The Thinking in Java Annotated Solution Guide*》電子文件中。僅需小額費用便可自 *www.BruceEckel.com.*網站取得。

1. 本章稍早題爲「優先序（precedence）」的那一節中，曾經提到兩行算式。請將它們放進程式中，證明它們所產生的結果並不相同。
2. 請將 ternary() 和 alternative() 這兩個函式放進程式中。
3. 請將標題爲 “if-else” 和 “return” 兩小節中的 test() 和 test2() 函式放進程式。
4. 撰寫一個程式，令它印出數值 1~100。
5. 修改上題，利用關鍵字 break，讓程式在印出數值 47 時終止。試著改用 return 辦到這一點。
6. 撰寫一個函式，使它接收兩個 String 引數，並運用各種 Boolean 比較動作來比較這兩個 String，印出比較結果。進行 == 和 != 比較動作的同時，也請執行 equals() 測試。請在 main() 之中使用不同的 String 物件來呼叫你所撰寫的函式。
7. 撰寫程式，產生 25 個 int 隨機數。針對每個數值，使用 if-then-else 述句來區分該值究竟大於、小於、等於下一個隨機數。
8. 修改上題，以「while 無限迴圈」包住你所撰寫的程式碼。程式持續執行，直到你以鍵盤動作（通常是按下 Control-C）中斷其執行。
9. 撰寫一個程式，以兩個巢狀的（nested）for 迴圈，和模數運算子（%）來偵測質數並列印出來。所謂質數（prime numbers）就是「除了自身和 1 之外，找不到任何數可以整除該數」的整數。

10. 設計 switch 述句，在每個 case 中皆列印訊息，並將此 switch 置於迴圈中，藉以測試每個 case。先在每個 case 之後擺上 break，然後移去 breaks，看看兩者有什麼差別。

4: 初始化和清理

Initialization & Cleanup

隨著電腦逐漸演化，「不安全」的編程手法逐漸成爲編程代價日益高漲的主因。

初始化（*initialization*）和清理（*cleanup*）正是眾多安全議題中的兩個。許多 C 程式的錯誤肇因於程式員疏於將變數初始化。尤其當他們使用程式庫時，如果不知道如何初始化一個程式庫組件（或其他必須被初始化的事物），更是如此。「清理」是個比較特殊的問題，因爲當某個元素被使用完畢，你很容易便忘了它的存在，畢竟它再也不會影響你。於是該元素依舊佔用寶貴的資源，而你可能很快便耗盡所有資源（尤其是記憶體）。

C++ 引入所謂「建構式（*constructor*）」觀念。建構式是一種特殊的函式，當物件被產生時，此式會被自動喚起。Java 也採納了建構式的觀念，並額外提供所謂的垃圾回收器（garbage collector）：當物件不再被使用，垃圾回收器能自動釋放該物件所佔用的記憶體資源。本章審視了物件初始化、清理的相關議題，並討論 Java 對此二者所提供的支援。

以建構式（constructor）確保初始化的進行

你可以這麼想：爲你所撰寫的每個 class 都發展一個 **initialize()**。名稱本身即有示意效果，表示它應該在物件被使用之前先被喚起。遺憾的是，這意謂使用者必須自己記得呼叫這個函式。在 Java 裡頭，class 的設計者可以透過「提供某個建構式（constructor）」的行爲，確保每個物件的初始化動作一定會被執行。如果某個 class 具備建構式，Java 便會在物件生成

之際，使用者有能力加以操作之前，自動呼叫其建構式，於是便能夠確保初始化動作一定被執行。

接下來令人頭痛的便是建構式的命名問題。這裡存在兩個問題，第一，你所使用的任何名稱都可能和 class 內的成員名稱衝突。第二，建構式由編譯器負責喚起，所以編譯器必須知道它將呼叫哪一個函式。C++ 的解法似乎是最簡單也最符合邏輯的，所以也被 Java 採用：建構式名稱和 class 名稱相同。對於這種「會在初始化動作進行時被呼叫」的函式而言，這種作法極爲合理。

以下便是一個帶有建構式的簡單 class：

```
//: c04:SimpleConstructor.java
// Demonstration of a simple constructor.

class Rock {
  Rock() { // This is the constructor
    System.out.println("Creating Rock");
  }
}

public class SimpleConstructor {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rock();
  }
} ///:~
```

現在，當物件生成之際：

```
new Rock();
```

編譯器會配置必要的儲存空間，並呼叫其建構式。你可以放心，在你得以操作此一物件之前，它絕對會被適當地初始化。

請注意，「每個函式的第一個字母以小寫表示」這種編程風格不適用於建構式身上，因爲建構式的名稱必須完全吻合 class 名稱。

和其它函式一樣，你也可以為建構式提供引數，藉以指定物件的生成方式。上例可以輕易做點修改，使其建構式接受一個引數：

```
//: c04:SimpleConstructor2.java
// Constructors can have arguments.

class Rock2 {
  Rock2(int i) {
    System.out.println(
      "Creating Rock number " + i);
  }
}

public class SimpleConstructor2 {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rock2(i);
  }
} ///:~
```

有了建構式引數，你得以為物件的初始化動作提供引數。假設 class **Tree** 擁有一個「接收單一整數」的建構式，此引數被用來表示樹的高度，那麼你便可以用這種方式來產生 **Tree** 物件：

```
Tree t = new Tree(12);  // 12-foot tree
```

如果 **Tree(int)** 是唯一建構式，編譯器便不允許你以任何其他方式來產生 **Tree** 物件。

建構式能夠消除極多問題，並使程式碼更易於閱讀。以前述程式碼片段為例，你不會看到任何程式碼直接呼叫某個「和 class 保持概念獨立」的 **initialize()**。在 Java 之中「定義」和「初始化」是一體的，兩者不可能彼此脫離而獨立存在。

建構式是一種很獨特的函式，因為它沒有回傳值。這和「回傳值為 **void**」有極大的差別。回傳 **void** 時，一般函式並不回傳任何東西，但是一般的函式能夠選擇是否要回傳些什麼東西。建構式則絕對不回傳任何東西，而且你也沒有任何選擇。如果它有一個回傳值，而且你有權力選擇你自己的回

傳型別（return type），編譯器勢必得透過某種方式來知道如何處理那個回傳值。

函式多載化（Method overloading）

任何程式語言中最重要的性質之一便是名稱的運用。當你產生某個物件，便是給予「對應的儲存空間」一個名稱。函式則是「行爲」的名稱。藉由名稱的運用來描述系統，你便可以發展出容易爲人理解（並修改）的程式。這和散文的書寫很相像 – 目標都是爲了和讀者溝通。

透過名稱，你可以取用任何物件和任何函式。精心挑選的名稱能夠幫助你自己和其他人更易了解程式碼的意涵。

不過，當我們將人類語言轉換爲程式語言時，問題隨之而生。通常同一個字可能會有不同的意義 – 它被多載化（*overloaded*）了。當這些不同意義之間的差別十分細微時，特別有用。你可以說「清洗（wash）這件襯衫」、「清洗（wash）這部車子」、以及「清洗（wash）這隻狗」。如果被迫說成「以清洗襯衫（shirtWash）的方式來清洗這件襯衫」、「以清洗車子（carWash）的方式來清洗這部車子」、「以清洗狗狗（dogWash）的方式來清洗這隻狗」，才能夠讓聆聽者得以不必區分所欲執行的動作之間的絲毫差異，那就顯得太過愚蠢了。大多數自然語言都有贅餘（redundant）特性，即使你漏掉少數幾個字，仍舊可以判斷出意思。我們不需要爲每個意義都提供個別名稱 – 我們可以從上下文（context）推論出實際意義。

大多數程式語言（尤其是 C）規定你必須爲每個函式提供獨一無二的識別字。所以你沒有辦法讓某個名爲 **print()** 的函式印出整數值，同時又讓另一個同樣名爲 **print()** 的函式印出浮點數值 – 每個函式都得有個獨一無二的名稱。

在 Java（和 C++）中，建構式是「函式名稱必須多載化」的另一個原因。由於建構式名稱是依據 class 名稱自動決定的，所以建構式名稱只能有一個。那麼如果你想以多種方式來產生物件，該當如何？假設你所開發的 class 能夠以標準方式將自己初始化，也可以從檔案中讀取資訊來初始化自己。於是你得有兩個建構式才行，其中一個不接收任何引數（此即所謂

default 建構式，或稱爲無引數（*no-arg*）建構式），另一個接收 **String** 引數，用以代表檔案名稱（檔案中儲存著物件的資料）。二者皆爲建構式，其名稱肯定相同 － 就是 class 的名稱。因此，爲了讓同名、具有不同引數型別的函式共同存在，多載化（overloading）便成了不可或缺的重要關鍵。多載化對於建構式來說是必然產物，而當它運用於一般函式時，一樣能帶來便利。

以下例子同時示範了「建構式」和「一般函式」的多載化：

```
//: c04:Overloading.java
// Demonstration of both constructor
// and ordinary method overloading.
import java.util.*;

class Tree {
  int height;
  Tree() {
    prt("Planting a seedling");
    height = 0;
  }
  Tree(int i) {
    prt("Creating new Tree that is "
        + i + " feet tall");
    height = i;
  }
  void info() {
    prt("Tree is " + height
        + " feet tall");
  }
  void info(String s) {
    prt(s + ": Tree is "
        + height + " feet tall");
  }
  static void prt(String s) {
    System.out.println(s);
  }
}

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");
    }
    // Overloaded constructor:
    new Tree();
  }
} ///:~
```

本例中的 **Tree** 物件有兩種生成形式：一種以幼苗形式來生成，不帶任何引數；一種以苗圃中的植物形式來生成，已有某個高度。爲了提供這種支援能力，我們供應兩個建構式：一個不接收任何引數（此稱爲 *default* 建構式[1]），另一個接收某個高度做爲引數。

你也許會想要以多種不同方式來呼叫 **info()**，例如帶有一個 **String** 引數，讓你印出額外訊息；或是不帶任何引數，讓你不印出任何額外訊息。對於這種概念明顯相同的事物，如果得賦予兩個不同的名稱，實在是很奇怪。幸運的是，method overloading（函式多載化）允許我們賦予這兩個函式相同的名稱。

區分多載化函式

如果多個函式具有相同名稱，Java 如何知道你要取用的究竟是哪一個呢？規則很簡單：每個多載化的函式都需具備獨一無二的引數列（*argument list*）。

稍加思考，你便會知道這極富意義。是的，除了引數列所帶來的差異，程式員還有什麼辦法可以區分兩個同名的函式？

即便是引數順序不同，都足以區分兩個函式（正常情況下你不會採用這種方式，因爲這麼做會讓程式難以維護）：

[1] Sun 所發表的某些 Java 文獻中，採用另一種笨拙但更具描述力的名稱：無引數（no-arg）建構式。但由於 *default* 建構式這個詞彙已行之多年，所以我延用之。

```java
//: c04:OverloadingOrder.java
// Overloading based on the order of
// the arguments.

public class OverloadingOrder {
  static void print(String s, int i) {
    System.out.println(
      "String: " + s +
      ", int: " + i);
  }
  static void print(int i, String s) {
    System.out.println(
      "int: " + i +
      ", String: " + s);
  }
  public static void main(String[] args) {
    print("String first", 11);
    print(99, "Int first");
  }
} ///:~
```

上例兩個 **print()** 都有相同的引數，但擺設順序不同，這使它們得以有所區別。

搭配基本型別（primitives）進行多載化

基本型別可以自動由較小型別晉升至較大型別。當它和多載機制搭配使用時，會產生某種輕微的混淆現象。以下程式說明當基本型別被置於多載化函式時，究竟會引發什麼現象：

```java
//: c04:PrimitiveOverloading.java
// Promotion of primitives and overloading.

public class PrimitiveOverloading {
  // boolean can't be automatically converted
  static void prt(String s)
    System.out.println(s);
  }

  void f1(char x) { prt("f1(char)"); }
```

```
  void f1(byte x) { prt("f1(byte)"); }
  void f1(short x) { prt("f1(short)"); }
  void f1(int x) { prt("f1(int)"); }
  void f1(long x) { prt("f1(long)"); }
  void f1(float x) { prt("f1(float)"); }
  void f1(double x) { prt("f1(double)"); }

  void f2(byte x) { prt("f2(byte)"); }
  void f2(short x) { prt("f2(short)"); }
  void f2(int x) { prt("f2(int)"); }
  void f2(long x) { prt("f2(long)"); }
  void f2(float x) { prt("f2(float)"); }
  void f2(double x) { prt("f2(double)"); }

  void f3(short x) { prt("f3(short)"); }
  void f3(int x) { prt("f3(int)"); }
  void f3(long x) { prt("f3(long)"); }
  void f3(float x) { prt("f3(float)"); }
  void f3(double x) { prt("f3(double)"); }

  void f4(int x) { prt("f4(int)"); }
  void f4(long x) { prt("f4(long)"); }
  void f4(float x) { prt("f4(float)"); }
  void f4(double x) { prt("f4(double)"); }

  void f5(long x) { prt("f5(long)"); }
  void f5(float x) { prt("f5(float)"); }
  void f5(double x) { prt("f5(double)"); }

  void f6(float x) { prt("f6(float)"); }
  void f6(double x) { prt("f6(double)"); }

  void f7(double x) { prt("f7(double)"); }

  void testConstVal() {
    prt("Testing with 5");
    f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5);
  }
  void testChar() {
    char x = 'x';
    prt("char argument:");
```

```
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testByte() {
    byte x = 0;
    prt("byte argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testShort() {
    short x = 0;
    prt("short argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testInt() {
    int x = 0;
    prt("int argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testLong() {
    long x = 0;
    prt("long argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testFloat() {
    float x = 0;
    prt("float argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testDouble() {
    double x = 0;
    prt("double argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  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();
  }
} ///:~
```

檢視本程式的輸出結果，你會發現常數值 5 被視為 **int**，所以如果某個多載化函式接受 **int** 做為引數，該函式便會被喚起。至於其他情況，如果你所提供的資料，其型別「小於」函式的引數，該資料的型別便會獲得晉升。**char** 便是如此，如果無法找到恰好吻合的 **char** 參數，它便會被晉升至 **int**。

如果你所提供的引數「大於」某個多載化函式所預期的引數時，又會引發什麼現象呢？修改上述程式，便能提供解答：

```
//: c04:Demotion.java
// Demotion of primitives and overloading.

public class Demotion {
  static void prt(String s)
    System.out.println(s);
  }

  void f1(char x) { prt("f1(char)"); }
  void f1(byte x) { prt("f1(byte)"); }
  void f1(short x) { prt("f1(short)"); }
  void f1(int x) { prt("f1(int)"); }
  void f1(long x) { prt("f1(long)"); }
  void f1(float x) { prt("f1(float)"); }
  void f1(double x) { prt("f1(double)"); }

  void f2(char x) { prt("f2(char)"); }
  void f2(byte x) { prt("f2(byte)"); }
  void f2(short x) { prt("f2(short)"); }
  void f2(int x) { prt("f2(int)"); }
  void f2(long x) { prt("f2(long)"); }
  void f2(float x) { prt("f2(float)"); }

  void f3(char x) { prt("f3(char)"); }
  void f3(byte x) { prt("f3(byte)"); }
  void f3(short x) { prt("f3(short)"); }
```

```
  void f3(int x) { prt("f3(int)"); }
  void f3(long x) { prt("f3(long)"); }

  void f4(char x) { prt("f4(char)"); }
  void f4(byte x) { prt("f4(byte)"); }
  void f4(short x) { prt("f4(short)"); }
  void f4(int x) { prt("f4(int)"); }

  void f5(char x) { prt("f5(char)"); }
  void f5(byte x) { prt("f5(byte)"); }
  void f5(short x) { prt("f5(short)"); }

  void f6(char x) { prt("f6(char)"); }
  void f6(byte x) { prt("f6(byte)"); }

  void f7(char x) { prt("f7(char)"); }

  void testDouble() {
    double x = 0;
    prt("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();
  }
} ///:~
```

此例之中，這些函式會接受較小的基本型別數值。如果你所提供的引數較大，你就得在小括號中指定型別名稱，做必要的轉型。如果少了這個動作，編譯器會送出錯誤訊息。

你應該明白，這是所謂的窄化轉型（*narrowing conversion*），表示在轉型過程中會遺失資訊。這也就是爲什麼編譯器強迫你一定要明白標示出來的原因，它要你自行負責後果。

以回傳值（return value）作為多載化的基準

如果你有以下懷疑，很正常：「爲什麼只能夠以 class 名稱和函式引數列做爲多載化的區分呢？爲什麼不依據函式的回傳值加以區分呢？」舉個例子，下面兩個函式有相同的名稱和相同的引數，但很容易區別：

```
void f() {}
int f() {}
```

是的，當編譯器可以很明確地從程式的前後狀態判斷出意義時，這麼做毫無問題，例如 **int x = f()**。但由於呼叫函式時可以不在乎其回傳值（這麼做通常是爲了只想獲得其「副作用」），所以如果你用這種方式來呼叫函式：

```
f();
```

Java 如何才能判斷應該喚起哪一個 **f()** 呢？程式碼的閱讀者又要如何才能明白這一點呢？基於此點，你無法以回傳型別做爲多載化函式的區分基準。

Default 建構式

一如先前所述，*default* 建構式（或「無引數建構式」）是一種不帶任何引數的建構式，被用於所謂「無特色物件」的生成。如果你所開發的 class 不具任何建構式，編譯器會自動爲你合成一個 *default* 建構式。例如：

```
//: c04:DefaultConstructor.java

class Bird {
  int i;
}

public class DefaultConstructor {
  public static void main(String[] args) {
    Bird nc = new Bird(); // default!
  }
} ///:~
```

其中的：

```
new Bird();
```

會產生一個新的物件，並呼叫 *default* 建構式 – 即使你沒有自行定義任何 *default* 建構式。如果沒有 *default* 建構式，上例就沒有任何函式可供呼叫，也就無法建構出我們想要的物件了。請注意，如果你自行定義任何一個建構式（不論有無引數），編譯器就不會爲你合成 *default* 建構式：

```
class Bush {
  Bush(int i) {}
  Bush(double d) {}
}
```

現在，如果你這麼寫：

```
new Bush();
```

編譯器會發出抱怨，因爲它無法找到吻合的建構式。當你沒有提供任何建構式，編譯器說：『你需要建構式，讓我爲你產生一個吧』。但如果你已經撰寫了某個建構式，編譯器會說：『你已經撰寫了建構式，顯然你知道你正在做什麼；如果你沒有撰寫 *default* 建構式，你必然是刻意的』。

關鍵字 this

假設你有兩個同型的物件，分別名爲 **a** 和 **b**。你可能想知道，究竟該以何種方式，才能分別透過這兩個物件呼叫 **f()**：

```
class Banana { void f(int i) { /* ... */ } }
Banana a = new Banana(), b = new Banana();
a.f(1);
b.f(2);
```

如果只有一個 **f()** ，那麼它又如何得知自己是被物件 **a** 或 **b** 呼叫呢？

爲了讓你能夠以十分便利的物件導向語法（也就是「送訊息給物件」這樣的意涵）來撰寫程式碼，編譯器暗自動了些手腳。事實的真象是，有個隱晦的第一引數被傳入 **f()**，此一引數便是「正被操作中的物件的 reference」。所以上述兩個函式的叫用形式其實變成這個樣子：

```
Banana.f(a,1);
Banana.f(b,2);
```

這種轉換只存於語言內部，你無法撰寫上面那樣的算式並順利通過編譯。不過，這麼寫可以讓你對實際發生的事情有些概念。

想像你現在正位於某個函式之內，你想取得當前的 object reference。由於該 reference 是編譯器偷偷傳入，所以並不存在其識別名稱。爲了這個原因，關鍵字 **this** 因應而生。這個關鍵字僅用於函式之內，能取得「喚起此一函式」的那個 object reference。你可以採取你面對任何 object reference 一樣的處理方式來處理這個 **this**。注意，如果你只是在某個 class 的某個函式內呼叫同一個 class 的另一個函式，沒有必要動用 **this**，直接呼叫即可，因爲編譯器會自動套用當前的 **this** reference。因此，你可以這麼寫：

```
class Apricot {
  void pick() { /* ... */ }
  void pit() { pick(); /* ... */ }
}
```

在 **pit()** 中，你可以將 **pick()** 寫成 **this.pick()**，但沒有必要如此。編譯器會自動爲你完成此事。當你必須明確指出當前的 object reference 究竟爲何時，才有需要動用關鍵字 **this**。例如，當你想要回傳目前的物件時，就需要在 **return** 述句中這麼寫：

```
//: c04:Leaf.java
// Simple use of the "this" keyword.

public class Leaf {
  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().increment().print();
```

```
  }
} ///:~
```

由於 **increment()** 透過關鍵字 **this** 回傳了當前的物件，所以我們可以輕易在同一個物件身上執行多次操作。

在建構式中呼叫建構式

當你為某個 class 撰寫多個建構式時，有時候你會想要在某個建構式中呼叫另一個建構式，以免撰寫重覆的程式碼。使用 **this** 關鍵字便可以做到這一點。

正常情況下當你寫下 **this**，指的是「此一物件」或「目前的物件」，並自動產生「目前的物件」的 reference。但是在建構式中，當你賦予 **this** 一個引數列，它有了不同的意義：它會呼叫符合該引數列的某個建構式。如此一來我們便能夠直接呼叫其他建構式（譯註：C++不允許如此）：

```
//: c04:Flower.java
// Calling constructors with "this."

public class Flower {
  int petalCount = 0;
  String s = new String("null");
  Flower(int petals) {
    petalCount = petals;
    System.out.println(
      "Constructor w/ int arg only, petalCount= "
      + petalCount);
  }
  Flower(String ss) {
    System.out.println(
      "Constructor w/ String arg only, s=" + ss);
    s = ss;
  }
  Flower(String s, int petals) {
    this(petals);
//!    this(s); // Can't call two!
```

```
    this.s = s; // Another use of "this"
    System.out.println("String & int args");
  }
  Flower() {
    this("hi", 47);
    System.out.println(
      "default constructor (no args)");
  }
  void print() {
//!    this(11); // Not inside non-constructor!
    System.out.println(
      "petalCount = " + petalCount + " s = "+ s);
  }
  public static void main(String[] args) {
    Flower x = new Flower();
    x.print();
  }
} ///:~
```

建構式 **Flower(String s, int petals)** 說明了一點;：雖然你能夠藉由 **this** 呼叫一個建構式，卻不能以相同手法呼叫兩個。此外，對另一建構式的呼叫動作必須置於最起始處，否則編譯器會發出錯誤訊息。

這個例子同時也展示了 **this** 的另一用途。由於引數 **s** 和資料成員 **s** 的名稱相同，會出現模稜兩可的情況。你可以寫下 **this.s**，明確表示你所取用的是資料成員 **s**。Java 程式碼中常能見到此種形式，本書亦常採用這種形式。

在 **print()** 中你看到了，編譯器不允許你在建構式以外的任何函式內呼叫建構式。

static 的意義

了解 **this** 之後，你應該比較能夠了解何謂 **static** 函式。它的意思是，對於這個函式，沒有所謂的 **this**。你無法在 **static** 函式中呼叫 non-**static** 函

式[2]（反向可行）。不過倒是可以經由 class 本身來呼叫其 **static** 函式，無需透過任何物件。事實上這也正是 **static** 函式存在的主要原因。它很像是 C 的全域函式。Java 沒有全域函式；將 **static** 函式置於 class 之中，可以提供「對其它 **static** 函式和 **static** 資料成員」的存取。

有些人認爲 **static** 函式不符合物件導向的精神，因爲它們具備了全域函式的語意；他們認爲，使用 **static** 函式時，並非以「送出訊息給物件」的方式來達成，因爲其間並不存在 **this**。此論頗爲公允，因此如果你發現自己動用大量 **static** 函式，你應該重新思考自己的策略。不過，**static** 是務實的產物，而且的確存在應用時機，所以它們是否歸屬於「物件導向編程的嚴格定義」內，應該留給理論家去辯論。即使 Smalltalk 也具備所謂 class 函式（等同於 **static** 函式）這樣的東西呢。

清理（Cleanup）：終結（finalization）與垃圾回收（garbage collection）

程式員能夠體會初始化的重要性，但常常遺忘清理的重要性。畢竟，有誰需要清理一個 **int** 呢？但是，使用程式庫時，在用完物件之後，如果只是將它棄而不顧，有時候會帶來危險。當然啦，Java 提供了垃圾回收器來回收那些不再被使用的物件的記憶體空間，但是請你想想某些不尋常的情況。假設你的物件並非經由 **new** 獲得某種「特殊」記憶體。垃圾回收器只知道釋放那些經由 **new** 配置得來的記憶體，因此它不知道如何釋放你這個物件所佔用的「特殊」記憶體。爲了因應這種情況，Java 提供一個 **finalize()** 函式，你可以爲自己的 class 定義此一函式。以下說明它被期望的運作方式。當垃圾回收器打算開始釋放你的物件所佔用的儲存空間時，會先呼叫其 **finalize()**，並且在下一次垃圾回收動作發生時才回收該物件

---

[2]除非你將某個 object reference 傳入 **static** method，然後透過該 reference（效用等同 **this**），你便可以呼叫 non-**static** methods 並存取 non-**static** fields 了。不過，如果你想從事諸如此類的動作，通常你只需製作普通的 non-**static** method 即可☺。

所佔用的記憶體。所以，如果你使用 **finalize()**，它便讓你得以在「垃圾回收時刻」執行某些重要的（你自己的）清理動作。

這中間有個潛伏的編程陷阱。某些程式員（尤其是 C++ 程式員）可能會先入爲主地認爲 **finalize()** 是 C++的解構式（*destructor*），也就是當物件被摧毀時會被自動喚起的函式。從這一點區分 C++ 和 Java 是很重要的，因爲 C++物件絕對會被摧毀（如果你的程式沒有臭蟲），而 Java 物件並不絕對會被垃圾回收器回收。換一個說法：

垃圾回收（garbage collection）不等於解構（destruction）

牢記這一點，你便可以遠離困擾。這句話的真義是，在你「永遠不再需要某個物件」之前，如果某些動作必得執行，你得自己動手執行它們。Java 並沒有解構式（*destructor*）或類似概念，所以你必須撰寫一個函式來執行清理動作。舉個例子，假設物件在產生過程中將自己繪製於螢幕上，如果你沒有自行動手將它從螢幕拭去，它便可能永遠不會被清理掉。如果你將某些擦拭影像的功能置於 **finalize()**，那麼當這個物件被垃圾回收器回收之前，其影像會先在螢幕上被拭去。如果物件沒有被回收，影像就持續存在。所以請牢記第二點：

你的物件可能不會被回收

你可能發現，某個物件所佔用的空間永遠沒有被釋放掉，因爲你的程式可能永遠不會逼近記憶體用完的那一刻。如果你的程式執行完畢，而垃圾回收器完全沒有被啓動以釋放你的物件所佔據的記憶體，那些空間便會在程式終止時才一次歸還給作業系統。這是好事，因爲垃圾回收器會帶來額外負擔，如果永遠不啓動它，就永遠不需要付出額外代價。

finalize() 存在是爲了什麼？

此刻，你也許會相信你不應該使用 **finalize()** 做一般用途的清理工作。那麼，它究竟爲何存在呢？

第三句金玉良言就是：

垃圾回收動作（garbage collection）只回收記憶體

也就是說，垃圾回收器存在的唯一理由，就是要回收那些在你程式中再也用不著的記憶體空間。所以任何和垃圾回收動作綑綁在一起的行爲，也都只能和記憶體或記憶體的釋放相關。

這是否意味如果你的物件內含其他物件，在 **finalize()** 中應該明確釋放那些物件？嗯，答案是 no，垃圾回收器所留意的是「物件記憶體」的釋放，不論物件被生成的方式究竟爲何。如此一來，對 **finalize()** 的需求便侷限於特定情況之下。這個情況就是：當你的物件以「物件生成」之外的方式配置了某種儲存空間。不過，你可能會觀察到，Java 的一切事物都是物件，那麼究竟怎樣才會碰到上述情況？

似乎只有在你透過 Java 的非正常管道來配置記憶體，打算做一些類似 C 會做的事情時，才是使用 **finalize()** 的適當時機。這種情況主要發生在採用原生函式（*native methods*）時，那是一種在 Java 程式中呼叫 non-Java 程式碼的方法（附錄 B 對此有所討論）。目前原生函式僅支援 C 和 C++ 兩個語言，不過由於這兩個語言可以再呼叫其他語言，所以實際上你可以呼叫所有語言。在 non-Java 程式碼中，C 的 **malloc()** 函式族系可能會被呼叫，用以配置儲存空間，而且除非你呼叫 **free()**，否則其佔用空間永遠不會被釋放，進而產生記憶體漏洞（*memory leak*）。當然啦，**free()** 是 C/C++ 函式，所以你得在 **finalize()** 中以原生函式加以呼叫。

閱讀至此，你或許明白，你不應該過度使用 **finalize()**。你是對的，它並不是擺放正常清理動作的合適地點。那麼，哪裡才是正常清理動作的執行場所呢？

你必須執行清理（cleanup）動作

爲了清理某個物件，使用者必須在「打算進行清理動作」時呼叫「用於清理任務」的函式。聽起來很簡單，但這麼做和 C++ 的解構式（destructor）觀念稍有抵觸。在 C++ 中，所有物件都會被摧毀，或說所

有物件都「應該」被摧毀。如果某個 C++ 物件以 local 形式生成（也就是佔用 stack 空間，這在 Java 不可能發生），那麼其解構動作會於該物件產生地點所在的大括號範圍結束前自動發生。如果該物件是以 **new** 產生（就像 Java 一樣），那麼其解構式會在程式員呼叫 C++ **delete** 運算子（Java 沒有這個運算子）時被喚起。如果 C++ 程式員忘了呼叫 **delete**，其解構式便永遠不會被喚起，並因此產生記憶體漏洞，而物件的其他部份也不會被清理。這類程式臭蟲很難追蹤。

相對之下，Java 不允許你產生 local 物件，你一定得使用 **new** 才行。不過 Java 並沒有 **delete** 運算子可供呼叫以釋放物件，因爲垃圾回收器會爲你釋放空間。所以，從簡化觀點來看，你可以說，因爲有了垃圾回收機制，所以 Java 不需要解構式。但當你隨著本書的內容前進，你會發現，垃圾回收器的存在無法消除對解構式的需求，也無法取代它的功用（而且由於你絕不應該直接呼叫 **finalize()**，所以 **finalize()** 不是解決此一問題的適當手段）。除了空間的釋放，如果你希望執行其他清理動作，你仍得自行呼叫恰當的函式，這個函式和 C++ 解構式等效，只不過少了點便利性。

finalize() 有個可以發揮實際效用的地方，就是用來觀察垃圾回收過程。以下範例爲你說明整個過程的進行，並摘要整理先前對垃圾回收動作的種種敘述：

```
//: c04:Garbage.java
// Demonstration of the garbage
// collector and finalization

class Chair {
  static boolean gcrun = false;
  static boolean f = false;
  static int created = 0;
  static int finalized = 0;
  int i;
  Chair() {
    i = ++created;
    if(created == 47)
      System.out.println("Created 47");
  }
```

```
  public void finalize() {
    if(!gcrun) {
      // The first time finalize() is called:
      gcrun = true;
      System.out.println(
        "Beginning to finalize after " +
        created + " Chairs have been created");
    }
    if(i == 47) {
      System.out.println(
        "Finalizing Chair #47, " +
        "Setting flag to stop Chair creation");
      f = true;
    }
    finalized++;
    if(finalized >= created)
      System.out.println(
        "All " + finalized + " finalized");
  }
}

public class Garbage {
  public static void main(String[] args) {
    // As long as the flag hasn't been set,
    // make Chairs and Strings:
    while(!Chair.f) {
      new Chair();
      new String("To take up space");
    }
    System.out.println(
      "After all Chairs have been created:\n" +
      "total created = " + Chair.created +
      ", total finalized = " + Chair.finalized);
    // Optional arguments force garbage
    // collection & finalization:
    if(args.length > 0) {
      if(args[0].equals("gc") ||
         args[0].equals("all")) {
        System.out.println("gc():");
        System.gc();
      }
```

```
      if(args[0].equals("finalize") ||
         args[0].equals("all")) {
        System.out.println("runFinalization():");
        System.runFinalization();
      }
    }
    System.out.println("bye!");
  }
} ///:~
```

上述程式產生許多 **Chair** 物件，並且在垃圾回收器啓動時間點上，停止產生 **Chair**s。由於垃圾回收器可能在任何時間啓動，你無法精確知道它究竟何時開始執行，因此程式中以名爲 **gcrun** 的旗標來標示垃圾回收器是否已經開始執行。另一個旗標 **f** 用來讓 **Chair** 通知 **main()** 中的迴圈，是否應該停止產生物件。這兩個旗標都應該在 **finalize()** 中加以設定。垃圾回收動作一旦發生，便會呼叫 **finalize()**。

另有兩個 **static** 變數：**created** 和 **finalized**，用來記錄 **Chair** 物件的數目以及被垃圾回收器終結的數目。每個 **Chair** 都有其自身的（non-**static**）**int i**，用以記錄自己的誕生序號。當 **Chair** 物件序號 47 被終結，旗標 **f** 會被設爲 **true**，停止 **Chair** 的生產。

所有事情都發生在 **main()** 的迴圈內：

```
    while(!Chair.f) {
      new Chair();
      new String("To take up space");
    }
```

你可能想知道，此一迴圈究竟如何結束，因爲迴圈中並沒有任何事物會更動 **Chair.f** 之值。喔，**finalize()** 會更動之，因爲它終究會回收至序號 47 的物件。

每次迭代所產生的 **String** 物件，只是爲了當做額外的配置空間，藉以激發垃圾回收器的啓動。因爲垃圾回收器會在它判斷「可用記憶體數量不足」時才開始執行。

欲執行此程式，你得經由命令列提供必要的引數："gc" 或 "finalize" 或 "all"。如果指定 "gc"，會令程式呼叫 **System.gc()**（強迫執行垃圾回收器）。如果指定 "finalize"，會呼叫 **System.runFinalization()**，理論上會讓每個未被終結（*finalized*）的物件都被終結。如果指定 "all"，會使得上述兩個函式都被呼叫。

這個程式所顯露出來的行爲，及其前一版（出現於本書第一版）的行爲，說明垃圾回收和終結的議題又有了新的發展。許多發展是關起門來秘密進行的。事實上在你閱讀本書之際，這個程式的行爲可能又再次有了變化。

如果 **System.gc()** 被喚起，終結動作便會發生於所有物件身上。這對早先的 JDK 而言，並非絕對必然 – 儘管文件上不是這麼說。此外，你也會觀察到，不論 **System.runFinalization()** 是否被喚起，似乎都沒有任何差別。

然而你會發現，只有當 **System.gc()** 被喚起於「所有物件被產生並被棄置」之後，所有的 *finalizer*s 才會被呼叫。如果你沒有呼叫 **System.gc()**，那麼只有某些物件會被終結。在 Java 1.1 中，**System.runFinalizersOnExit()** 用來讓程式結束時得以執行所有的 *finalizer*s。不過，此一設計最後被認爲有問題，所以這個函式便被宣告爲不再適用。這條線索曝露出 Java 設計者仍在反覆研究並企圖解決垃圾回收和終結的議題。我們希望 Java 2 中一切事情能夠獲得好的結果。

先前的程式說明了，「*finalizer*s 絕對會被執行」這項保證是千真萬確的。但是只有在你強迫它發生的時候才會如此。如果你沒有讓 **System.gc()** 被喚起，你會得到如下輸出結果：

```
Created 47
Beginning to finalize after 3486 Chairs have been
created
Finalizing Chair #47, Setting flag to stop Chair
creation
After all Chairs have been created:
total created = 3881, total finalized = 2684
bye!
```

因此並非所有的 *finalizers* 在程式執行完畢時都被喚起。如果 **System.gc()** 曾經被呼叫，它就會終結、摧毀當時不再被使用的所有物件。

請記住，無論是垃圾回收動作或終結動作，都不保證一定會發生。如果 Java 虛擬機器（JVM）並未面臨記憶體耗盡的情境，它不會浪費時間於垃圾回收上，這很聰明。

死亡條件（The death condition）

一般情況下，你的程式不能倚靠「**finalize()** 被呼叫」才正常運作。你得發展個別的「清理」函式，並自行呼叫之。所以，看起來，**finalize()** 只能在大多數程式員都不會用到的「無名記憶體」清理動作上發揮作用。不過，只要你的程式不倚靠 **finalize()** 的被呼叫，那麼它還有一個很有意思的用途，那就是物件的「死亡條件（*death condition*）[3]」的檢查。

當你對某個物件不再感興趣 – 也就是它可以被清理時 – 這個物件應該處於某種狀態，使它所佔用的記憶體空間得以被安全釋放。舉例來說，如果這個物件代表某個已開啓的檔案，那麼該檔案在此一物件被回收前，應該先被關閉。只要這個物件的任何部份未被適當清理，你的程式便存在難以被找出的臭蟲。**finalize()** 的價值便在於它可以用來找出這種情況 – 雖然它並不一定會被喚起。如果由於某個終結（*finalized*）動作的發展，使得這個程式臭蟲曝光，你便可以找出問題所在。這正是我們最關注的事情。

以下是個簡單例子，示範可能的使用方式，：

```
//: c04:DeathCondition.java
// Using finalize() to detect an object that
// hasn't been properly cleaned up.

class Book {
```

[3] 這是我和 Bill Venners（www.artima.com）共同開授的研討班中，由他所創的一個詞彙。

```
  boolean checkedOut = false;
  Book(boolean checkOut)
    checkedOut = checkOut;
  }
  void checkIn() {
    checkedOut = false;
  }
  public void finalize() {
    if(checkedOut)
      System.out.println("Error: checked out");
  }
}

public class DeathCondition {
  public static void main(String[] args) {
    Book novel = new Book(true);
    // Proper cleanup:
    novel.checkIn();
    // Drop the reference, forget to clean up:
    new Book(true);
    // Force garbage collection & finalization:
    System.gc();
  }
} ///:~
```

在這個例子中，死亡條件是「所有 **Book** 物件都被預期在它們被回收之前先被登錄（checked in）」。但在 **main()** 中，由於程式員的錯誤，有一本書未被登錄。如果沒有 **finalize()** 來檢查死亡條件，這會是一隻棘手的臭蟲。

請記住，**System.gc()** 被用來強迫終結動作的發生（你應該在程式開發過程中進行此事，以便協助偵錯）。不過，即使沒有使用到它，在反覆執行此程式的情況下（假設程式會配置夠多的儲存空間，使得垃圾回收器啟動），最終還是有可能找出遺失的 **Book**。

垃圾回收器（garbage collector）的運作方式

如果你以往所用的程式語言，從 heap 空間配置物件的代價十分高昂，那麼你可能直覺地假設，Java 這種從 heap 空間配置所有物件（基本型別除

外）的體制，也得付出高昂成本。然而垃圾回收器對於物件的生成速度，卻可能帶來顯著的加速效果。乍聽之下令人覺得奇怪 — 儲存空間的釋放竟然會影響儲存空間的配置 — 但這的確是某些 JVMs（Java 虛擬機器）的運作方式，這同時也意味，Java 從 heap 空間配置物件的速度，能夠逼近其他語言自 stack 挖掘空間的速度。

舉個例子，你可以將 C++ heap 想像是一塊場地，在其中，每個物件不斷監視屬於自己的地盤。它們可能在稍後某個時刻不再繼續佔用目前所佔用的空間，這些空間因此必須被重新運用。在某些 JVMs 裡頭，Java heap 和上述觀念有著顯著的不同；比較像是輸送帶，每當你配置新的物件，這條履帶便往前移動。這意味物件空間的配置速度非常快。「heap 指標」只是很單純地往前移動至未經配置的區域，因此其效率和 C++ stack 配置動作不相上下。當然，相關的記錄動作還是得付出額外代價，不過不需要進行諸如「搜尋可用空間」之類的大動作。

現在，你也許會注意到，heap 實際上不是個輸送帶，如果你只用這種方式來處理 heap，最後勢必遭遇極爲頻繁的記憶體分頁置換動作（*paging*），因而大幅影響系統效能，並在不久之後耗盡資源。成功的關鍵在於垃圾回收器的介入。回收垃圾的同時，它會重新安排 heap 內的所有物件，使它們緊密排列。這樣一來便將「heap 指標」移至更靠近輸送帶的前端，避免分頁失誤（*page fault*）的發生。垃圾回收器會進行事物重新排列，使得高速、無限自由 heap 空間的模式有機會於空間配置時實現。

你得多多了解各種垃圾回收器（GC）的架構，才能更加明白上述方式如何運作。參用計數（*reference counting*）是一種很單純、速度很慢的 GC 技術：每個物件皆內含所謂的參用計數器，每當某個 reference 連接至某個物件，其計數器便累加 1。每當 reference 離開其生存範圍，或被設爲 **null**，其計數器便減 1。因此，reference counts 的管理動作很簡單，但程式執行過程中必須持續付出額外負擔，因爲垃圾回收器必須逐一走訪一長串的物件，並且在發現某個物件的參用計數器爲零時，釋放其所佔用的空間。這麼做的缺點之一是，如果物件彼此之間形成循環性的交互參考，那麼即便整群物件都已成爲垃圾，它們仍舊可能有著非零的 reference counts。找出這種自我參考的整群物件，對垃圾回收器來說是件極度龐大的工程。Reference counting 常常被拿來說明某種類型的垃圾回收機制，但似乎從來沒有被真正用於任何 JVM 身上。

爲求更快的速度，垃圾回收動作並非以 reference counting 方式爲之。它所依據的基本理念是，能夠根據「存活於 stack（堆疊）或 static storage（靜態儲存空間）上」的 reference 而追蹤到的物件，才算是生命尚存的物件。整個追蹤鏈可能會涵括好幾層物件。因此如果你從 stack 或 static storage 出發，走訪所有 references 之後，便能找到所有存活的物件。針對每個你所找到的 reference，你都必須再鑽進它所代表（指向）的物件，然後循著該物件內含的所有 references，再鑽入它們所指的物件。如此反覆進行，直到訪遍「根源於 stack 或 static storage 上」的 reference 所形成的整個網絡爲止。你所走訪的每個物件都必定是存活的。請記住，這種作法並不存在「自我引用群」（self-referential groups）的問題 – 那些物件不會被找到，自然而然變成了垃圾。

在這種方式下，JVM 使用所謂的「自省式（*adaptive*）」垃圾回收機制。至於它如何處理它所找到的存活物件，視 JVM 採用哪種變形而定。眾多變形之中有一種作法名爲 *stop-and-copy*（停止而後複製）。就像字面所顯示的意義一樣，它先將執行中的程式暫停下來（所以它並非一種於背景執行的垃圾回收機制），然後將所有找到的物件從原本的 heap 複製到另一個 heap，並將所有垃圾捨棄不顧。當物件被複製到新 heap，係以頭尾相連的方式排列，於是新的 heap 排列緊湊，並因此在配置新儲存空間時得以簡單地從最末端騰出空間來，一如先前所描述。

當然，當物件從某處被移至另一處，所有指向它的那些 references 都必須獲得修正。「來自 heap 或 static storage」的 references 可被立即修正，但可能還有其他「指向此一物件」的 references，得透過後繼走訪過程才能找到。找到它們之後才能調整其值（你可以想像有個表格，將舊位址映射至新位址）。

對於這種所謂的「複製式回收器（*copy collectors*）」而言，兩個問題會造成效率低落。首先，你得有兩個 heaps，然後你得在兩個獨立的 heaps 之間將記憶體內容來回搬動。這麼做得維護實際所需記憶體數量的兩倍。某些 JVMs 對此問題的處理方式是，從 heap 配置出所需的 chunks（譯註：大塊大塊記憶體），然後將資料從 chunk 複製到另一個 chunk。

第二個問題在於複製。你的程式進入穩定狀態之後，可能只會產生少量垃圾，甚至不產生垃圾。儘管如此，複製式回收器仍然會將所有記憶體自某處複製到另一處，這麼做極度浪費氣力。爲了避免這種情形發生，有些 JVMs 會偵測是否沒有新垃圾產生，並因此轉換到另一種機制（此即所謂「自省式（adaptive）」的字面意義）。有一種名爲 *mark and sweep*（標示而後清理）的機制爲早期的 Sun JVM 採用。對一般用途而言，這種方式其實速度頗慢，但是當你知道你只產生少量垃圾甚至沒有產生垃圾時，它的速度就很快了。

mark and sweep 依循的運作規則，同樣是從 stack 和 static storage 出發，追蹤走訪所有 references，進而找出所有存活的物件。每當它找到一個存活物件，便設定該物件內的旗標，加以標示。此時物件尚未被回收。當整個標示程序都完成了，清理動作才會開始。清理過程中，不復存活的物件會被釋放（譯註：因爲這些物件所佔用的空間被釋放掉了，所以 heap 中的被使用空間呈現不連續狀態）。沒有任何複製動作發生。所以如果收集器決定將「使用狀態呈現不連續」的 heap 加以密集（compact），它得重新整理它所找到的物件。

stop-and-copy 這個名稱，意味這種垃圾回收動作並非執行於背景；相反的，GC 啓動同時，執行中的程式會被暫停。你可以在 Sun 文獻中發現，許多參考文獻將垃圾回收視爲低優先權的背景程序，但事實上 GC 並非以此種方式實作，至少早期的 Sun JVM 並非如此。當記憶體可用數量趨低時，Sun 垃圾回收器才會啓動，而 *mark and sweep* 方式也必須在程式停止的情況下才能運作。

如前文所述，此處所描述的 JVM，記憶體配置單位是一大區塊（blocks，譯註：而非只是如物件般大小的小塊記憶體）。如果你配置了大型物件，它會擁有專屬區塊。周密的 *stop-and-copy* 作法，必須在你釋放舊有物件之前，先將所有存活物件從舊 heap 複製到新 heap，這個過程需要搬動大量記憶體。如果以區塊（blocks）方式爲之，GC 可以在回收過程中使用廢棄區塊（dead blocks）來複製物件。每個區塊都有世代計數（*generation count*），記錄它是否還存活。正常情況下只有上次 GC 運作之後所產生的區塊才會被密集（compact）起來；如果某個區塊尚被某處引用，其世代計數會有所記錄。這種方式可以處理數量極大而生命短暫的暫時物件。完整的清理動作會定期出現 — 大型物件仍然不會被複製（只是其世代計數會

受到影響）。內含小型物件的那些區塊則被複製並密集。JVM 會監督 GC 效率，如果發現 GC 因爲「所有物件皆長期存活」而變得效率不彰，它會轉換至 *mark and sweep* 模式。同樣道理，JVM 會追蹤 *mark and sweep* 模式的績效，如果 heap 斷裂情況太嚴重，它會轉回 *stop-and-copy* 模式。此即「自省式（adaptive）」的由來。我們可以這麼說：*stop-and-copy* 和 *mark and sweep* 兩者是自省的、互生的。

JVM 可以加上許多額外的技巧以提昇速度。載入器（loader）和即時（Just-In-Time，JIT）編譯器是其中格外重要者。一旦某個 class 必須被載入（通常是在你爲它產生第一個物件時），編譯器會先找到其 **.class** 檔案位置，然後將 class byte codes 載入記憶體。這時候有一種手法，就是對所有程式碼進行 JIT 動作，但這麼做有兩個缺點：它會花費更多時間，這些時間散落在程式執行過程的許多地點；這麼做也會增加執行碼佔用空間（byte codes 佔用的空間比 JIT 展開後的程式碼小得多），因而可能導致分頁置換（paging）動作發生，這絕對會拖累程式的執行速度。另一種方法則是所謂的「緩式評估（*lazy evaluation*）」，意思是程式碼只有在需要用到時才透過 JIT 編譯。因此如果某段程式碼永遠不被執行，也就永遠不會被加以 JIT 編譯。

成員初始化（Member initialization）

Java 保證，變數絕對會在它們被使用之前被適當初始化。當變數被定義於函式之內，Java 會運用編譯期錯誤訊息來貫徹它的保證。所以如果你這麼寫：

```
void f() {
  int i;
  i++;
}
```

你會得到錯誤訊息，告訴你 **i** 可能沒有被初始化。當然，編譯器可以給 **i** 一個預設值，但這畢竟很有可能是程式員的疏失，那麼即便提供預設值也沒有實質效果。強迫程式員提供初值，更有可能找到程式臭蟲。

如果某個 class 的資料成員隸屬基本型別，情況就稍有不同。因爲所有函式都可以初始化或使用該資料值，所以強迫使用者一定得在使用之前給定適當初值是不切實際的。不過，放任它持有毫無意義的初值也很危險，所以每個「隸屬基本型別」的 class 資料成員都保證一定有初值。下面例子顯示這些初值：

```
//: c04:InitialValues.java
// Shows default initial values.

class Measurement {
  boolean t;
  char c;
  byte b;
  short s;
  int i;
  long l;
  float f;
  double d;
  void print() {
    System.out.println(
      "Data type      Initial value\n" +
      "boolean        " + t + "\n" +
      "char           [" + c + "] "+ (int)c +"\n"+
      "byte           " + b + "\n" +
      "short          " + s + "\n" +
      "int            " + i + "\n" +
      "long           " + l + "\n" +
      "float          " + f + "\n" +
      "double         " + d);
  }
}

public class InitialValues {
  public static void main(String[] args) {
    Measurement d = new Measurement();
    d.print();
    /* In this case you could also say:
```

```
    new Measurement().print();
    */
  }
} ///:~
```

程式輸出如下：

```
Data type       Initial value
boolean         false
char            [ ] 0
byte            0
short           0
int             0
long            0
float           0.0
double          0.0
```

char 之值為零，所以印出空白。

稍後你會看到，當你在 class 之內定義了一個 object reference 而卻沒有為它設定初值（指向某個物件），該 reference 會獲得一個特殊值 **null**（這是 Java 關鍵字）。

從輸出結果可以觀察到，即使沒有指定任何初值，它們都會被自動初始化。所以至少我們不必冒著「拿無初值變數來使用」的風險。

指定初值

如果你想為某個變數設定初值，應該怎麼進行？最直接的方式便是直接在 class 的變數定義處指定其值。請注意，C++ 不允許你這麼做 — 雖然 C++ 初學者總是這麼嘗試☺。下面的例子改變了 class **Measurement** 中的欄位定義，藉以提供初值：

```
class Measurement {
  boolean b = true;
  char c = 'x';
  byte B = 47;
  short s = 0xff;
  int i = 999;
  long l = 1;
```

```
  float f = 3.14f;
  double d = 3.14159;
  //. . .
```

你也可以使用同樣的手法來爲基本型別以外的物件設初值。如果 **Depth** 是個 class，你可以在加入 **Depth** 變數的同時給予初值，像這樣：

```
class Measurement {
  Depth o = new Depth();
  boolean b = true;
  // . . .
```

如果你在尚未爲 **o** 提供初值前便嘗試使用它，會得到所謂的異常（exception，第 10 章討論），這是一種執行期錯誤。

你甚至可以呼叫某個函式來提供初值：

```
class CInit {
  int i = f();
  //...
}
```

當然，函式可以擁有引數，但這些引數不可以是尚未初始化的其他 class 成員。因此，你可以這麼寫：

```
class CInit {
  int i = f();
  int j = g(i);
  //...
}
```

卻無法這麼寫：

```
class CInit {
  int j = g(i);
  int i = f();
  //...
}
```

編譯器會在進行前置參考（forward referencing）時適當指出問題所在，因爲上述程式的正確性顯然相依於初始化順序，無法通過編譯。

此種初始化方式極簡單又直覺，不過有個限制：每個 **Measurement** 物件都有相同的初值。有時候這是你要的，有時候你需要更彈性的作法。

以建構式（Constructor）進行初始化動作

建構式可用來執行初始化動作，因而賦予你更多編程上的彈性。這是因爲你可以在執行期呼叫函式並執行某些動作以決定初值。不過，使用這種作法時，有件事必須牢記在心：你無法杜絕發生於建構式執行之前的自動初始化動作。所以，如果你這麼寫：

```
class Counter {
  int i;
  Counter() { i = 7; }
  // . . .
```

i 值會先被初始化爲 0，然後才變成 7。任何基本型別和 object references 皆如此，就連那些定義時便給定初值的變數也不例外。基於這個理由，編譯器不會強迫你一定要在建構式內的某個特定地點將元素初始化，或是在你使用它們之前先初始化，因爲初始化動作已經是一種保證[4]。

初始化次序

class 中的初始化次序取決於變數在 class 中的定義次序。變數定義也許會散落各處，而且有可能介於各個函式定義之間。但所有變數一定會在任何一個函式（甚至是建構式）被呼叫之前完成初始化。舉個例子：

```
//: c04:OrderOfInitialization.java
// Demonstrates initialization order.
```

[4] 與此相較，C++ 具有所謂的「建構式初值列（*constructor initializer list*）」，能夠讓初始化動作發生於建構式主體之前，並適用於所有物件身上。請參考《*Thinking in C++*》第二版（內含於本書所附光碟，亦可於 *www.BruceEckel.com* 獲得）。

```
// When the constructor is called to create a
// Tag object, you'll see a message:
class Tag {
  Tag(int marker) {
    System.out.println("Tag(" + marker + ")");
  }
}

class Card {
  Tag t1 = new Tag(1); // Before constructor
  Card() {
    // Indicate we're in the constructor:
    System.out.println("Card()");
    t3 = new Tag(33); // Reinitialize t3
  }
  Tag t2 = new Tag(2); // After constructor
  void f() {
    System.out.println("f()");
  }
  Tag t3 = new Tag(3); // At end
}

public class OrderOfInitialization {
  public static void main(String[] args) {
    Card t = new Card();
    t.f(); // Shows that construction is done
  }
} ///:~
```

我在 **Card** 之中刻意將數個 **Tag** 物件的定義散落四處，藉以證明它們的初始化動作的確會在進入建構式或任何其他函式之前發生。此外 **t3** 會在建構式內被重新設值。輸出結果如下：

```
Tag(1)
Tag(2)
Tag(3)
Card()
Tag(33)
f()
```

因此，**t3** 這個 reference 會被初始化兩遍，一次在呼叫建構式之前，一次在呼叫建構式之後。第一次所產生的物件會被丟棄，可能稍後會被當成垃圾回收掉。乍見之下這似乎不太有效率，但這麼做可以保證得到妥當的初始化。是的，如果我們定義了某個多載化建構式，其中並未爲 **t3** 設初值，而 **t3** 定義處又沒有提供預設初值，會發生什麼不妙的事呢？

靜態資料（static data）的初始化

當資料是 **static** 型式，情況沒有什麼不同：如果資料隸屬基本型別而你又沒有加以初始化，它便會被設爲基本型別的標準初值；如果它是某個 object reference，初值便是 **null** — 除非你產生新物件並將這個 reference 指向它。

如果想要在定義處設定初值，作法和面對 non-**static**s 時沒有兩樣。注意，不論產生多少物件，**static** 變數都只佔用一份儲存空間。因此當我們打算將 **static** 儲存空間加以初始化，便會發生問題。下面這個例子淸楚突顯出這一問題：

```
//: c04:StaticInitialization.java
// Specifying initial values in a
// class definition.

class Bowl {
  Bowl(int marker) {
    System.out.println("Bowl(" + marker + ")");
  }
  void f(int marker) {
    System.out.println("f(" + marker + ")");
  }
}

class Table {
  static Bowl b1 = new Bowl(1);
  Table() {
    System.out.println("Table()");
    b2.f(1);
  }
  void f2(int marker) {
    System.out.println("f2(" + marker + ")");
```

```
  }
  static Bowl b2 = new Bowl(2);
}

class Cupboard {
  Bowl b3 = new Bowl(3);
  static Bowl b4 = new Bowl(4);
  Cupboard() {
    System.out.println("Cupboard()");
    b4.f(2);
  }
  void f3(int marker) {
    System.out.println("f3(" + marker + ")");
  }
  static Bowl b5 = new Bowl(5);
}

public class StaticInitialization {
  public static void main(String[] args) {
    System.out.println(
      "Creating new Cupboard() in main");
    new Cupboard();
    System.out.println(
      "Creating new Cupboard() in main");
    new Cupboard();
    t2.f2(1);
    t3.f3(1);
  }
  static Table t2 = new Table();
  static Cupboard t3 = new Cupboard();
} ///:~
```

Bowl 讓你得以觀看 class 的誕生過程，**Table** 和 **Cupboard** 會在它們自身的 class 定義式中產生 **Bowl static** 成員。請記住，**Cupboard** 會在 **static** 定義之前先產生 non-**static Bowl b3**。輸出結果說明了實際發生之事：

```
Bowl(1)
Bowl(2)
Table()
```

```
f(1)
Bowl(4)
Bowl(5)
Bowl(3)
Cupboard()
f(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()
f(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()
f(2)
f2(1)
f3(1)
```

static 的初始化動作只在必要時刻才會發生。如果你沒有產生 **Table** 物件，也沒有取用 **Table.b1** 或 **Table.b2**，那麼 **static Bowl b1** 和 **b2** 就永遠不會被產生。不過它們的初始化動作只會發生在第一個 **Table** 物件被產生之際，也就是在第一個 **static** 的存取動作發生之際。自此之後，**static** 物件便不會再被初始化。

如果 **static**s 並未因爲早先的物件生成過程而被初始化，那麼初始化次序會以 **static**s 爲優先，然後才是 non-**static** 物件。這一現象可從輸出結果觀察得到。

爲物件生成過程做一份摘要整理，應該很有用。讓我們考慮一個名爲 **Dog** 的 class：

1. 當某個型別爲 **Dog** 的物件首次被產生出來，或是當 class **Dog** 的 **static** 函式或 **static** 資料成員首次被存取，Java 直譯器必須搜尋環境變數 classpath 所指定的位置，找出 **Dog.class**。

2. 一旦 **Dog.class** 被載入（稍後你會學到，這樣就是產生一個 **Class** 物件），它的所有 **static** 初始動作會被執行起來。因此 **static** 初始化動作僅會發生一次，就是在 **Class** 物件首次被載入時。

3. 當你 **new Dog()**，建構過程會先爲 **Dog** 物件在 heap 上配置足夠的儲存空間。

4. 這塊儲存空間會先被清爲零，並自動將 **Dog** 物件內所有隸屬基本型別的欄位設爲預設值（對數字來說就是零，對 **boolean** 和 **char** 來說亦同），並將 references 設爲 **null**。

5. 執行所有出現於欄位定義處的初始化動作。

6. 執行建構式。就如你將在第六章所見，這中間可能會牽扯極多動作，尤其當繼承關係捲入時。

static 明確初始化
Explicit static initialization

Java 允許你將多個 **static** 初始化動作組織起來，置於特殊的「**static** 建構子句（有時也稱爲 *static block*）」中，看起來像這樣：

```
class Spoon {
  static int i;
  static {
    i = 47;
  }
  // . . .
```

看起來像是個函式，但其實是在關鍵字 **static** 之後緊接著函式本體。這樣的程式碼就像其他形式的 **static** 初始化動作一樣，只會被執行一次：在你首次產生 class 物件或首次存取該 class 的 **static** 成員（即使你從未產生過該 class 物件）時。例如：

```
//: c04:ExplicitStatic.java
// Explicit static initialization
// with the "static" clause.

class Cup {
  Cup(int marker) {
    System.out.println("Cup(" + marker + ")");
  }
```

```
  void f(int marker) {
    System.out.println("f(" + marker + ")");
  }
}

class Cups {
  static Cup c1;
  static Cup c2;
  static {
    c1 = new Cup(1);
    c2 = new Cup(2);
  }
  Cups() {
    System.out.println("Cups()");
  }
}

public class ExplicitStatic {
  public static void main(String[] args) {
    System.out.println("Inside main()");
    Cups.c1.f(99);  // (1)
  }
  // static Cups x = new Cups();  // (2)
  // static Cups y = new Cups();  // (2)
} ///:~
```

Cups 的「**static** 初始化動作」將於 **static** 物件 **c1** 被存取之際執行，也就是發生在標示 (1) 那行。如果標示 (1) 的那一行被註解掉，而標示 (2) 的那些程式碼被解除註解，那麼「**static** 初始化動作」就會發生在標示 (2) 的程式行上。如果 (1) 和 (2) 皆被註解掉，**Cup**s 的 **static** 初始化動作便永遠不會發生。標示 (2) 的程式行中如果有一行或兩行未被註解掉，其實沒有區別，因為 **static** 初始化動作只會執行一次。

Non-static 實體（instance）初始化動作

Java 也為物件內的 non-**static** 變數的初始化行為提供了類似語法。以下便是一例：

```
//: c04:Mugs.java
// Java "Instance Initialization."
```

```
class Mug {
  Mug(int marker) {
    System.out.println("Mug(" + marker + ")");
  }
  void f(int marker) {
    System.out.println("f(" + marker + ")");
  }
}

public class Mugs {
  Mug c1;
  Mug c2;
  {
    c1 = new Mug(1);
    c2 = new Mug(2);
    System.out.println("c1 & c2 initialized");
  }
  Mugs() {
    System.out.println("Mugs()");
  }
  public static void main(String[] args) {
    System.out.println("Inside main()");
    Mugs x = new Mugs();
  }
} ///:~
```

其中你可以看到實體（instance）初始化子句：

```
  {
    c1 = new Mug(1);
    c2 = new Mug(2);
    System.out.println("c1 & c2 initialized");
  }
```

看起來和 **static** 初始化子句一模一樣，只不過少了關鍵字 **static**。如果我們想提供 *anonymous inner class*（無名的內隱類別，詳見第八章）的初始化動作，此語法不可或缺。

Array 的初始化

在 C 裡頭爲 array 設立初值是一件容易出錯而且令人生厭的工作。C++使用 *aggregate initialization*（聚合初始化）讓這個動作更安全[5]。Java 沒有像 C++一般的所謂「aggregates（聚合物）」，因爲 Java 裡頭的每樣東西都是物件。Java 也有 array，也支援 array 初始化功能。

array 其實不過是一連串物件或一連串基本資料，它們都必須同型，並以一個識別字（名稱）封裝在一起。array 的定義與使用，係以中括號做爲「索引運算子（*indexing operator*）**[ ]**」。定義一個 array 很簡單，只要在型別名稱之後接著一組空白中括號即可：

```
int[] a1;
```

你也可以將中括號置於識別字之後，意義完全相同：

```
int a1[];
```

這種寫法符合 C 和 C++ 程式員的預期。不過前一種風格或許更合情理，因爲它表示其型別是個 **int** array。本書將採用此種風格。

編譯器並不允許你告訴它究竟 array 有多大。此一行爲把我們帶回 reference 議題。此刻你所擁有的，只是一個 reference，代表某個 array，但是並沒有對應空間。如果想爲該 array 產生必要的儲存空間，你得撰寫初始化算式才行。對 array 而言，初始化動作可以出現在程式的任何地點，但你也可以使用一種特殊的初始化算式，它僅能出現於 array 生成處。此一特殊初始化形式是由成對大括號所括住的一組值來設定。這種情況下，儲存空間的配置（等同於以 **new** 來進行）由編譯器負責。例如：

```
int[] a1 = { 1, 2, 3, 4, 5 };
```

那麼，什麼情況使你只想定義 array reference 而不定義 array 本身呢：

---

[5] 關於 C++ aggregate initialization，詳見《*Thinking in C++*》第二版。

```
int[] a2;
```

唔，Java 允許你將某個 array 指派給另一個 array，所以你可以這麼寫：

```
a2 = a1;
```

你所做的其實不過是 reference 的複製罷了，以下是個明證：

```
//: c04:Arrays.java
// Arrays of primitives.

public class Arrays {
  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]++;
    for(int i = 0; i < a1.length; i++)
      System.out.println(
        "a1[" + i + "] = " + a1[i]);
  }
} ///:~
```

你看到了，**a1** 被給定初值，**a2** 卻沒有；稍後我們將 **a1** 指派給 **a2**。

這裡顯示了一些新觀念：任何 array（無論組成份子是物件或基本型別資料）都有一個固有成員，你可以查詢其值（但無法加以改變），藉以得知這個 array 所含的元素個數。此一成員即是 **length**。由於 Java array 和 C/C++ array 都一樣從 0 開始對元素計數，所以你能使用的最大索引值便是 **length-1**。如果索引值超過界限，C 和 C++會毫無異議地接受，因此你得以存取所有記憶體內容。許多惡名昭彰的程式臭蟲正是源之於此。Java 不同，它會對你的越界存取動作發出執行期錯誤（亦即發出異常 *exception*，詳見第 10 章），使你免於遭受此類問題。當然，每次存取 array 時所做的邊界檢查，會耗費時間成本、帶來額外的程式碼，而且你無法關閉此一功能。這意味在關鍵時機上，array 的存取可能會是程式效率不彰的因素。但是在考量 Internet 安全性和程式員生產力之後，Java 設計者認爲值得付出這個代價。（譯註：有一種被稱爲 buffer overflow 的攻擊方式，便是利用「存取非法記憶體位址」的手法達到入侵系統的目的）

如果撰寫程式時你完全不知道你的 array 需要多少元素，情形又該如何？喔，只要以 **new** 來產生元素即可。下面這個例子，使用 **new** 沒有問題，即使它所產生的乃是基本型別的 array（注意，**new** 無法產生 non-array 的基本型別資料）：

```
//: c04:ArrayNew.java
// Creating arrays with new.
import java.util.*;

public class ArrayNew {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod + 1;
  }
  public static void main(String[] args) {
    int[] a;
    a = new int[pRand(20)];
    System.out.println(
      "length of a = " + a.length);
    for(int i = 0; i < a.length; i++)
      System.out.println(
        "a[" + i + "] = " + a[i]);
  }
} ///:~
```

由於 array 的大小係透過 **pRand()** 隨機選取而來，所以 array 的生成毫無疑問發生於執行期。此外你可以從本例輸出結果觀察到，基本型別的 array 所含的元素會被自動初始化爲「空」值：對數值和 **char** 而言爲 0，對 **boolean** 而言爲 **false**。

當然，你也可以在同一述句中完成 array 的定義和初始化：

```
int[] a = new int[pRand(20)];
```

如果你要處理的是基本型別以外的 objects array，那麼你一定得使用 **new** 來生成各個元素。下面這個例子中，reference 相關議題再度出現，因爲你所產生的其實只是由 references 組成的 array（而非內含實際物件）。以 **Integer** wrapper type（外覆型別）爲例，它是個 class 而非基本型別：

```
//: c04:ArrayClassObj.java
```

```
// Creating an array of nonprimitive objects.
import java.util.*;

public class ArrayClassObj {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod + 1;
  }
  public static void main(String[] args) {
    Integer[] a = new Integer[pRand(20)];
    System.out.println(
      "length of a = " + a.length);
    for(int i = 0; i < a.length; i++) {
      a[i] = new Integer(pRand(500));
      System.out.println(
        "a[" + i + "] = " + a[i]);
    }
  }
} ///:~
```

在這個例子中，即使以 **new** 產出 array 之後：

```
Integer[] a = new Integer[pRand(20)];
```

這還是個 references array。只有在「生成新的 **Integer** 物件以初始化 reference」之後，初始化動作才算完成：

```
a[i] = new Integer(pRand(500));
```

如果你忘記產生物件，當你企圖讀取空的 array 位置時，會獲得一個異常（exception）。

請留意 **print** 述句中 **String** 物件的形成。你看到了，指向 **Integer** 物件的那個 reference 被自動轉型，變成物件內容的 **String** 表示式。

我們也可以將一串初值置於成對大括號中，以此來對 objects array 設初值。這種作法有兩種形式：

```
//: c04:ArrayInit.java
// Array initialization.

public class ArrayInit {
```

```
  public static void main(String[] args) {
    Integer[] a = {
      new Integer(1),
      new Integer(2),
      new Integer(3),
    };

    Integer[] b = new Integer[] {
      new Integer(1),
      new Integer(2),
      new Integer(3),
    };
  }
} ///:~
```

有時候這種寫法很有用，但因爲 array 的容量得在編譯期決定，所以其用途很受限制。初值列最後的逗號可有可無（這對冗長的初值列的維護頗有幫助。）

array 的第二種初始化形式提供了十分便利的語法，可產生和「C 的可變引數列」（*variable argument lists*，在 C 中稱爲 "varargs"）相同的效應，可以含括未定個數的引數和未定的型別。由於所有 classes 都繼承自共同根源 class **Object**（循著本書腳步，慢慢你會對此了解更多），所以你可以撰寫某個函式，令它接受一個 **Object** array，然後像下面這樣加以呼叫：

```
//: c04:VarArgs.java
// Using the array syntax to create
// variable argument lists.

class A { int i; }

public class VarArgs {
  static void f(Object[] x) {
    for(int i = 0; i < x.length; i++)
      System.out.println(x[i]);
  }
  public static void main(String[] args) {
    f(new Object[]
        new Integer(47), new VarArgs(),
        new Float(3.14), new Double(11.11) });
```

```
    f(new Object[] {"one", "two", "three" });
    f(new Object[] {new A(), new A(), new A()});
  }
} ///:~
```

此刻，你無法在這些未知的物件身上進行太多動作。這個程式使用 **String** 自動轉換，使每個 **Object** 得以從事一些有用的事情。第 12 章會涵蓋所謂的「動態時期型別鑑識（*run-time type identification*，RTTI）」機制，你將學會如何發覺此類物件的確切型別，使你得以在其身上從事一些更有趣的活動。

多維度（Multidimensional）arrays

Java 讓你得以輕鬆產生多維度的 arrays:

```
//: c04:MultiDimArray.java
// Creating multidimensional arrays.
import java.util.*;

public class MultiDimArray {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod + 1;
  }
  static void prt(String s) {
    System.out.println(s);
  }
  public static void main(String[] args) {
    int[][] a1 = {
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
    for(int i = 0; i < a1.length; i++)
      for(int j = 0; j < a1[i].length; j++)
        prt("a1[" + i + "][" + j +
            "] = " + a1[i][j]);
    // 3-D array with fixed length:
    int[][][] a2 = new int[2][2][4];
    for(int i = 0; i < a2.length; i++)
      for(int j = 0; j < a2[i].length; j++)
        for(int k = 0; k < a2[i][j].length;
```

```
          k++)
        prt("a2[" + i + "][" +
            j + "][" + k +
            "] = " + a2[i][j][k]);
    // 3-D array with varied-length vectors:
    int[][][] a3 = new int[pRand(7)][][];
    for(int i = 0; i < a3.length; i++) {
      a3[i] = new int[pRand(5)][];
      for(int j = 0; j < a3[i].length; j++)
        a3[i][j] = new int[pRand(5)];
    }
    for(int i = 0; i < a3.length; i++)
      for(int j = 0; j < a3[i].length; j++)
        for(int k = 0; k < a3[i][j].length;
            k++)
          prt("a3[" + i + "][" +
              j + "][" + k +
              "] = " + a3[i][j][k]);
    // Array of nonprimitive objects:
    Integer[][] a4 = {
      { new Integer(1), new Integer(2)},
      { new Integer(3), new Integer(4)},
      { new Integer(5), new Integer(6)},
    };
    for(int i = 0; i < a4.length; i++)
      for(int j = 0; j < a4[i].length; j++)
        prt("a4[" + i + "][" + j +
            "] = " + a4[i][j]);
    Integer[][] a5;
    a5 = new Integer[3][];
    for(int i = 0; i < a5.length; i++) {
      a5[i] = new Integer[3];
      for(int j = 0; j < a5[i].length; j++)
        a5[i][j] = new Integer(i*j);
    }
    for(int i = 0; i < a5.length; i++)
      for(int j = 0; j < a5[i].length; j++)
        prt("a5[" + i + "][" + j +
            "] = " + a5[i][j]);
  }
} ///:~
```

以上程式碼列印時用了 **length**，所以程式碼不受限於特定 array 大小。

第一個例子說明多維的 primitives（基本型別的）array。你可以用成對大括號來劃分 array 中的諸向量：

```
    int[][] a1 = {
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
```

每一組中括號都代表 array 的不同層級。

第二個例子示範以 **new** 配置一個三維 array。在這裡，整個 array 的配置動作一次完成：

```
int[][][] a2 = new int[2][2][4];
```

第三個例子示範「array 之中形成整個矩陣的那些向量可以是任意長度」（譯註：也就是說 Java 的二維 array 不見得是矩形，可以是三角形；三維 array 也不見得是長方體）

```
    int[][][] a3 = new int[pRand(7)][][];
    for(int i = 0; i < a3.length; i++) {
      a3[i] = new int[pRand(5)][];
      for(int j = 0; j < a3[i].length; j++)
        a3[i][j] = new int[pRand(5)];
    }
```

其中第一個 **new** 生成一個 array，其第一維元素長度由亂數決定，剩餘部份未知。**for** 迴圈中的第二個 **new** 會填滿第二維元素。直到執行第三個 **new**，第三個索引值才告確定。

你可以從輸出結果觀察到，如果沒有明確給定 array 的元素值，它們會被自動初始化爲零。

你可以使用類似手法處理非基本型別的 objects array。第四個例子便是用來示範這一點。此例同時也說明了，透過成對大括號，將多個 **new** 算式組織在一起的能力：

```
    Integer[][] a4 = {
```

```
    { new Integer(1), new Integer(2)},
    { new Integer(3), new Integer(4)},
    { new Integer(5), new Integer(6)},
  };
```

第五個例子示範如何一步一步打造非基本型別的 objects array：

```
  Integer[][] a5;
  a5 = new Integer[3][];
  for(int i = 0; i < a5.length; i++) {
    a5[i] = new Integer[3];
    for(int j = 0; j < a5[i].length; j++)
      a5[i][j] = new Integer(i*j);
  }
```

其中的 **i*j** 只是爲了將某個有趣數值置於 **Integer** 中。

摘要

諸如建構式（constructor）這樣精巧複雜的初始化機制，應該能夠讓你體會到，程式的初始化行爲有多麼重要。Stroustru 當初設計 C++ 時，對於生產力所做的第一項觀察便是：有極大比例的編程問題來自於對變數的不適當初始化。這類臭蟲很難找出來。相同問題也源自不適當的清理行爲。由於建構式讓你得以保證執行起適當的初始化動作和清理動作（編譯器不允許你在未呼叫適當建構式之前產生物件），所以你可以取得完全的控制，也可以獲得安全保障。

在 C++ 中，解構動作格外重要，因爲經由 **new** 所產生的物件必須被明確加以摧毀。但由於 Java 的垃圾回收器會自動釋放無用的物件所佔據的記憶體空間，因此 Java 裡頭大多數時候並不需要同等效果（作爲清理之用）的函式。在那些不需要解構式的場合，Java 垃圾回收器大幅簡化了編程手續，而且在記憶體管理上增加了更爲迫切的安全性。某些垃圾回收器甚至可以清理其他如繪圖資源和 file handles 之類的資源。不過，垃圾回收器也增加了執行期成本。付出的代價很難清楚描述，因爲本書撰寫之際，整個 Java 直譯器到處充斥著緩慢因子。如果這點能夠改變，我們便有能力判斷

垃圾回收器所帶來的額外負擔是否真的不利於「將 Java 應用於一般類型的程式身上」。（一個大問題便是：垃圾回收器具有不可預測的特性）

由於所有的物件都保證會被建構，所以和建構式有關的事物相當多，實際上多過本章所提的說明。具體而言，當你運用複合技術（*composition*）或繼承技術（*inheritance*）來產生新 classes，前述的建構保證依舊成立，但這時候得有某些額外語法加入，以便提供支援。你會在接下來的章節中學到複合和繼承，以及它們對建構式產生的影響。

練習

某些經過挑選的題目，其解答置於《*The Thinking in Java Annotated Solution Guide*》電子文件中。僅需小額費用便可自 *www.BruceEckel.com*.網站取得。

1. 撰寫一個帶有 *default* 建構式（亦即不接收任何引數）的 class，並在其中列印某些訊息。請爲此 class 產生一些物件。

2. 請爲上題中的 class 增加一個多載化建構式，令它接收一個 **String** 引數，並印出你所提供的訊息。

3. 請爲題 2 的 class 產生一個 object references array，但不需實際產生物件來附著於 array。執行此程式時，請注意建構式中的初始化訊息有沒有被列印出來。

4. 繼續題 3 的內容，產生一些物件，將它們附著於 references array。

5. 產生 **String** objects array，並爲每個元素指派一個字串。以 **for** 迴圈將所有內容印出來。

6. 產生一個名爲 **Dog** 的 class，具有多載化的 **bark()**。此一函式應根據不同的基本資料型別進行多載化，並根據被呼叫的版本，印出不同類型的狗吠（barking）、咆哮（howling）等訊息。撰寫 **main()** 來呼叫所有不同版本。

7. 修改題 6 程式，讓兩個多載化函式都有兩個引數（型別不同），但二者順序恰好相反。請檢驗其運作方式。

8. 撰寫不帶建構式的 class，並在 **main()** 中產生其物件，驗證所謂 *default* 建構式是否真的被編譯器自動合成出來。

9. 撰寫具有兩個函式的 class，在第一個函式內呼叫第二個函式兩次：第一次呼叫時不使用 **this**，第二次呼叫時使用 **this**。

10. 撰寫具有兩個（多載化）建構式的 class，並在第一個建構式中透過 **this** 呼叫第二個建構式。

11. 撰寫具有 **finalize()** 的 class，並在其中列印訊息。請在 **main()** 中針對該 class 產生一個物件。試解釋這個程式的行爲。

12. 修改題 11 的程式，讓你的 **finalize()** 絕對會被呼叫。

13. 撰寫名爲 **Tank** 的 class，此 class 的狀態可以是滿的（filled）或空的（emptied）。其死亡條件（*death condition*）是：物件被清理時必須處於空狀態。請撰寫 **finalize()** 以檢驗死亡條件是否成立。請在 **main()** 中測試 **Tank** 可能發生的幾種使用方式。

14. 撰寫一個 class，內含未初始化的 **int** 和 **char**，印出其值以檢驗 Java 的預設初始化動作。

15. 撰寫一個 class，內含未初始化的 **String** reference。證明這個 reference 會被 Java 初始爲 **null**。

16. 撰寫一個 class，內含一個 **String** 欄位，在定義處初始化，另一個 **String** 欄位由建構式初始化。這兩種方法有什麼分別？

17. 撰寫一個 class，擁有兩個 **static String** 欄位，其中一個在定義處初始化，另一個在 **static** block 中初始化。現在，加入一個 **static** 函式用以印出兩個欄位值。請證明它們都會在被使用之前完成初始化動作。

18. 撰寫一個 class，內含 **String** 欄位，並採用「instance（實體）初始化」方式。試描述此一性質的用途（請提出一個和本書所言不同的用途）。

19. 撰寫一個函式，能夠產生二維的 **double** array 並加以初始化。array 的容量由函式的引數決定，其初值必須落在函式引數所指定的上下限之間。撰寫第二個函式，印出第一個函式所產生的 array。試著在 **main()** 中透過這兩個函式產生不同容量的 array，並列印其內容。

20. 重複題 19，但改爲三維 array。

21. 將本章中的 **ExplicitStatic.java** 內部標示 (1) 的程式行註解起來，並驗證「靜態初始化子句」並未被喚起。然後再將標示爲 (2) 的其中一行程式碼解除註解，並驗證「靜態初始化子句」的確被呼叫了。最後再將標示爲 (2) 的另一行程式碼解除註解，並驗證「靜態初始化子句」只會執行一次。

22. 在 **Garbage.java** 實驗中，以三個不同的引數 "gc"、"finalize"、"all" 分別執行該程式。重複多次，看看你是否能從輸出結果中看出什麼固定模式。接下來請改變程式碼，讓 **System.runFinalization()** 在 **System.gc()** 之前被呼叫，並觀察結果。

5: 隱藏實作細目

Hiding the Implementation

讓變動的事物與不變的事物彼此隔離，是物件導向設計（OOD）的首要考量。

這對程式庫（library）來說尤其重要。程式庫必須搏得其使用者（即客戶端程式員，*client programmer*）的信賴。程式員必須確認，即使程式庫釋出了新版本，他們亦無需改寫程式碼。從另一方面說，程式庫開發者在不影響客端程式員的程式碼的情況下，必須擁有修改與強化的自由。

只要透過約定，便可達成上述目的。例如，更動程式庫內的 classes 時，程式庫開發者必須遵守「不刪去任何既有函式」的約定，因爲這麼做將使客端程式碼無法正常運作。另一部份更爲棘手：就資料成員而言，程式庫開發者如何才能夠知道，究竟哪些資料成員已被客端程式員取用？對那些「僅是 class 實作細目中的一部份，因而不想讓客端程式員直接使用」的函式來說，情形亦同。如果程式庫開發者想捨棄舊有的實作方式，採用全新實作細目，情況又當如何？要知道，對那些成員的任何更動，都可能造成客端程式碼無法正常運作。程式庫開發者將因此動彈不得，無法改變（強化）任何東西。

爲了解決上述問題，Java 提供了存取權限飾詞（*access specifiers*），讓程式庫開發者得以指明哪些成員可供客端程式員取用。存取權限控制等級，從最寬鬆到最嚴格，依序爲 **public**、**protected**、friendly（不指定關鍵字）以及 **private**。依據前一段文字，你可能會認爲，做爲一個程式庫設計者，你會希望儘可能讓每個成員都是 **private**，而且僅對外公開你希望客端程式員使用的成員。完全正確！當然啦，這對那些使用其他程式語言（尤其是 C）並且毫無限制取用每樣事物的人來說，十分違反直覺。但是當本章結束，你應該不會對 Java 提供的「存取權限控制」的價值再有任何懷疑。

不過，組件程式庫（components library）的概念，以及「誰有資格取用其中組件」等概念都尚未完備。組件究竟應該如何結合，以形成具有內聚力的程式庫單元，至今尚有許多問題還待解決。Java 係透過關鍵字 **package** 加以控制，而存取權限會受到「class 是否位於同一個 package，或位於另一個 package」的影響。因此本章一開始，你會先學習如何將程式庫組件置入 packages 中，接下來便能夠學習「存取權限飾詞」的全面意義。

package:程式庫單元（library unit）

當你使用關鍵字 **import** 匯入整個程式庫，例如：

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

你取用的便是所謂的 package。這種寫法會將 Java 標準公用程式庫（utility library，也就是 **java.util**）整個引入。舉個例子，class **ArrayList** 位於 **java.util** 內，所以你可以選擇指定其全名 **java.util.ArrayList**（如此便無需動用上述的 **import** 述句），或是在寫過 **import** 述句之後，簡短寫成 **ArrayList**。

如果你只想引入單一 class，也可以在 **import** 述句中指定該 class 名稱：

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

那麼，不需要在 **ArrayList** 之前另加飾詞便可直接使用（譯註：這裡所謂飾詞是指 **java.util**，用來指定 package 名稱）。不過，這麼一來就法直接使用 **java.util** 內的其他 classes。

此類匯入（importing）動作的存在理由，是爲了提供命名空間（name spaces）的管理機制。所有 class 成員名稱皆被相互隔絕。class **A** 的函式 **f()**，其名稱不會和 class **B** 中具有相同標記式（signature）的 **f()** 相衝突。但是 class 名稱衝突的問題又要如何解決呢？如果你所開發的 **stack** class，被安裝在某一台機器，而其上已裝有他人撰寫之 **stack** class，不就遇上名稱衝突的問題了嗎？由於 Java 被用於網際網絡，在 Java 程式執行過程中，classes 會被自動下載，所以在使用者完全不知道的情況下這還是有可能發生的。

這一類可能發生的名稱衝突問題，正是 Java 必須全面掌控命名空間的原因，也正是 Java 必須不受 internet 特質限制而有能力產生獨一無二命名的重要原因。

截至目前，本書大多數範例都存於單一檔案中，並設計用於本機（local）端，因而尚未遭遇 package 命名困擾。這種情況下，class 的名稱被置於 default package 中。這當然也是一種選擇，而且基於簡化原則，即便到了本書末尾，仍有可能採用這種方式。不過當你希望你所開發的 Java 程式能和同一台機器上的其他 Java 程式和平共處時，你便得思考如何杜絕 class 的名稱衝突。

當你產生 Java 原始檔，此檔案通常被稱爲編譯單元（*compilation unit*）或轉譯單元（*translation unit*）。每個編譯單元的名稱皆需以 **.java** 作結，其中只能有一個與檔案同名的 **public** class（大小寫納入考慮，但不含括副檔名 **.java**）。每個編譯單元只能有一個 **public** class，否則編譯器不接受。package 之外的世界無法看見該編譯單元內的其餘 classes（如果有的話），這些 classes 主要用來爲那個主要的 **public** class 提供支援。

當你編譯 **.java** 檔，你所得到的輸出檔案，名稱恰與 .java 檔中的每一個 class 相同，只不過多了副檔名 **.class**。因此，數量較少的 **.java** 檔案編譯後，能夠得到數量較多的 **.class** 檔。如果你曾有編譯式語言的使用經驗，也許已經習慣透過編譯器產生所謂中間形式檔（通常是 .obj 檔），再透過連結器（linker）或程式庫產生器（librarian），將同爲中間形式的檔案結合起來。但這並非 Java 的運作方式。Java 的可執行程式乃是一組 **.class** 檔。Java 的 **jar** 壓縮工具能將眾多 **.class** 檔結合起來並予以壓縮。

Java 直譯器（interpreter）負責這些檔案的搜尋、載入、解譯[1]。

所謂程式庫，其實就是一組 class 檔。其中每個檔案都有一個 **public** class（非強迫，但通常如此），所以每個檔案都是一個組件（components）。如果你希望這些組件隸屬同一個群組，便可使用關鍵字 **package**。

當你在檔案起始處這麼寫（注意，**package** 述句必須是檔案中註解以外的第一行程式碼）：

```
package mypackage;
```

便會讓此編譯單元成為 **mypackage** 程式庫中的一部份。或是說，這麼做會讓此一編譯單元內的 **public** class，被置於 **mypackage** 這個名稱的保護傘下。任何人如果想使用該 class，必須指定全名，或者使用關鍵字 **import** 搭配 **mypackage**。請注意，Java package 的命名習慣是全部採用小寫字母，即使居中字詞也不例外。

假設上例的檔案名為 **MyClass.java**。這意謂其中可以有（且僅能有）唯一一個 **public** class，且其名稱一定得是 **MyClass**（注意大小寫）：

```
package mypackage;
public class MyClass {
  // . . .
```

現在，如果有人想使用 **MyClass**，或 **mypackage** 內的其他 **public** classes，他們必須使用關鍵字 **import**，方能運用 **mypackage** 內的其他名稱，否則便得指定完整名稱。

```
mypackage.MyClass m = new mypackage.MyClass();
```

[1] Java 並不強迫你一定得用直譯器。有些 Java 原生碼編譯器（native-code compiler），能產生單一可執行檔。（譯註：原生碼編譯器產生出來的可執行檔，將失去跨平台執行能力）

關鍵字 **import** 可以造成比較簡潔的效果：

```
import mypackage.*;
// . . .
MyClass m = new MyClass();
```

身爲程式庫的設計者，你應該明白，關鍵字 **package** 和 **import** 所提供的，乃是將單一全域命名空間加以切割，使得無論多少人使用 internet，無論多少人撰寫 Java classes，都不會發生命名衝突問題。

獨一無二的 package 命名

你可能已經觀察到，由於 package 並非真的將 .class 檔包裝成單一檔案，而是可能由許多 **.class** 檔組成，這麼一來便有可能造成混亂。爲了杜絕混亂，一個符合邏輯的作法是將 package 中的所有 **.class** 檔置於單一目錄下。也就是透過作業系統階層性的檔案結構來解決。這是一種解決方式，稍後介紹 **jar** 工具時，你會見到另一種解決方式。

將 package 涵括的所有檔案都置於同一磁碟目錄下，必須先解決另外兩個問題：如何產生獨一無二的 package 名稱，以及如何找出可能藏於目錄結構某處的 classes。正如第二章所說，將 **.class** 檔案所在的路徑位置編寫成 **package** 名稱，便解決了上述問題。編譯器會強迫這樣做，不過習慣上我們會以 class 開發者的 internet 域名（的相反順序）做爲 **package** 名稱的第一部份。由於域名絕對獨一無二，所以如果你依循此一習慣，便保證你所命名的 package 絕對獨一無二，不會有名稱上的衝突（除非你將域名轉讓給他人，而對方使用你過去所使用的相同路徑名稱來撰寫 Java 程式碼）。當然，如果你沒有自己的域名，就得自行產生一組不可能與他人重複的 package 名稱。如果你打算公開釋出自己的 Java 程式碼，付出一點點代價取得一個專屬域名，是相當值得的。

上述技巧的第二部份，便是將 **package** 名稱分解爲你的機器上的磁碟目錄名稱。於是每當 Java 程式執行並需要載入 **.class** 檔時（一旦程式有必要產生 class 物件，或首次存取 class **static** 成員時便會動態進行之），便可以找出 **.class** 檔案所在的目錄位置。

Java 直譯器（interpreter）的處理方式如下。首先找出環境變數 CLASSPATH（此乃透過作業系統加以設定。Java 或 Java 相關工具的安裝程式也可能爲你設定）。CLASSPATH 含有一個或多個目錄，每個目錄被視爲 **.class** 檔的搜尋起點。Java 會從這個起點開始，並將 **package** 名稱中的每個句點替換爲斜線（於是 **package foo.bar.baz** 變成 **foo\bar\baz** 或 **foo/bar/baz** — 依作業系統而有不同），以獲得在 CLASSPATH 起點下的路徑名稱。得出的路徑會接續於 CLASSPATH 的各個項目之下（譯註：不同的項目以分隔字元區分之，每個項目都代表一個磁碟目錄。分隔字元隨作業系統而有不同）。這些路徑名稱便是直譯器搜尋你所產生的 **.class** 的地點。直譯器也會搜尋它自己所在位置下的某些標準目錄。

以我的域名 **bruceeckel.com** 爲例，我來做一些更清楚的解說。將我的域名反轉得到 **com.bruceeckel**，這便成了我所開發的 classes 的一個舉世唯一的名稱。.com、edu、org 等名稱原本在 Java package 中是大寫的。不過 Java 2 之後，package 的完全名稱已經改爲全部小寫。假設我決定產生一個名爲 **simple** 的程式庫，於是我更進一步細分此一名稱，獲得的 package 名稱是：

```
package com.bruceeckel.simple;
```

現在，這個 package 名稱便可在下列兩個檔案中，做爲命名空間保護傘：

```
//: com:bruceeckel:simple:Vector.java
// Creating a package.
package com.bruceeckel.simple;

public class Vector {
  public Vector() {
    System.out.println(
      "com.bruceeckel.util.Vector");
  }
} ///:~
```

當你開發自有的 packages 時你會發現，**package** 述句必須是檔案中註解以外的第一行程式碼。第二個檔案看起來極爲相似：

```
//: com:bruceeckel:simple:List.java
// Creating a package.
package com.bruceeckel.simple;

public class List {
  public List() {
    System.out.println(
      "com.bruceeckel.util.List");
  }
} ///:~
```

上述兩個檔案皆被我置於我的系統上的同一個子目錄下：

```
C:\DOC\JavaT\com\bruceeckel\simple
```

如果你沿著此一目錄位置往回看，你會看到 package 的名稱 **com.bruceeckel.simple**，可是路徑名稱前的那一部份呢？那一部份由 CLASSPATH 環境變數負責，在我的機器上其值爲：

```
CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT
```

你看到了，CLASSPATH 之中可以含括多個不同的搜尋路徑。

當我們改用 JAR 檔時，情形又有不同。你得將 JAR 檔名（而不僅只是它所在的位置）於 CLASSPATH 環境變數中註明清楚。所以，對 **grape.jar** 來說，你的 CLASSPATH 應該含括：

```
CLASSPATH=.;D:\JAVA\LIB;C:\flavors\grape.jar
```

妥善設定 CLASSPATH 之後，以下程式檔可置於任何目錄之下而正常運作：

```
//: c05:LibTest.java
// Uses the library.
import com.bruceeckel.simple.*;

public class LibTest {
```

```
  public static void main(String[] args) {
    Vector v = new Vector();
    List l = new List();
  }
} ///:~
```

當編譯器面對 **import** 述句時，它會先搜尋 CLASSPATH 所指定的目錄，檢視子目錄 com\bruceeckel\simple 下的內容，找出檔名吻合的已編譯檔（對 **Vector** 來說是 **Vector.class**，對 **List** 來說是 **List.class**）。請注意，**Vector**、**List** classes 以及其中欲被使用的函式，都必須是 **public**。

CLASSPATH 的設定，對 Java 初學者而言是一件棘手的事（至少當初對我而言確是如此）。所以 Sun 讓 Java 2 的 JDK 更聰明一些。你會發現，在你安裝之後，即使完全沒有設定 CLASSPATH，你仍然能夠編譯基本的 Java 程式，並且加以執行。不過，如果想編譯並執行本書所附的一套原始碼（可從本書所附光碟或於 *www.BruceEckel.com* 網站取得），你仍須稍加修改 CLASSPATH（本套原始碼中亦有相關說明）。

衝突

如果兩個程式庫皆以 * 形式匯入，而且具有相同名稱，會發生什麼事情？例如，假設有個程式這麼寫：

```
import com.bruceeckel.simple.*;
import java.util.*;
```

由於 **java.util.*** 也含有 **Vector** class，所以這可能導致衝突。不過，只要你沒有真的寫出實際引發衝突的程式碼，一切都會相安無事 – 這是好事，不然你可能會得鍵入許多字才能阻止所有衝突發生。

但如果你現在嘗試產生一個 **Vector**，便會引發衝突：

```
Vector v = new Vector();
```

此行所取用的究竟是那個 **Vector** class？編譯器無從得知，讀者亦無從得知。所以編譯器會發出錯誤訊息，強迫你明確指出其名稱。如果我想使用的是標準的 Java **Vector**，我就得這麼寫：

```
java.util.Vector v = new java.util.Vector();
```

這種寫法（配合 CLASSPATH）能夠完全指出該 **Vector** 的所在位置。除非我還需要另外使用 **java.util** 中的其他東西，否則無需寫出述句 **import java.util.***。

自訂一個程式庫

有了以上認識，現在你可以開發自己專屬的工具程式庫來降低或消除重複的程式碼。例如下面這個 class 能夠產生 **System.out.println()** 的一個別名，減少打字負擔。這個 class 可做爲 **tools** package 的一部份：

```
//: com:bruceeckel:tools:P.java
// The P.rint & P.rintln shorthand.
package com.bruceeckel.tools;

public class P {
  public static void rint(String s) {
    System.out.print(s);
  }
  public static void rintln(String s) {
    System.out.println(s);
  }
} ///:~
```

你可以使用這個便捷的工具來列印 **String**，無論需要換行（**P.rintln()**）或是不需換行（**P.rint()**）。

你大概可以猜出，此檔所在的目錄位置，必定以 CLASSPATH 中的某個目錄爲首，然後接續 **com/bruceeckel/tools**。編譯之後，**P.class** 便可透過 **import** 述句被任何一個程式使用：

```
//: c05:ToolTest.java
// Uses the tools library.
import com.bruceeckel.tools.*;

public class ToolTest {
  public static void main(String[] args) {
    P.rintln("Available from now on!");
    P.rintln("" + 100); // Force it to be a String
```

```
    P.rintln("" + 100L);
    P.rintln("" + 3.14159);
  }
} ///:~
```

請注意，只要置於 **String** 算式之中，所有物件都可輕易被強迫轉成 **String** 形式；上例以空的 **String** 爲首的算式，正是這種手法。這引出了另一個有趣的觀察：如果你呼叫 **System.out.println(100)**，它並不會將 100 轉成 **String**。由於某種額外的重載（overloading）行爲，你可以讓 **P** class 有這樣的表現（這是本章末尾的習題之一）。

從現在開始，當你完成某個有用的工具程式，便可把它加至 **tools** 目錄中，或是你自己私人的 **util** 或 **tools** 目錄中。

利用 imports 改變行為

Java 並不具備 C 的「條件編譯（*conditional compilation*）」功能。此功能讓你得以切換開關，令程式產生不同行爲，無需更動程式碼。Java 拿掉這個功能的原因，可能是因爲此功能在 C 語言中多半被用來解決跨平台問題，也就是根據不同的平台編譯程式碼中的不同部份。由於 Java 本身的設計可自動跨越不同平台，所以應該不需要此功能。

不過條件編譯仍具有其他實用價值。除錯便是常見的用途之一：在開發過程中開啓除錯功能，在出貨產品中關閉除錯功能。Allen Holub（*www.holub.com*）提出了以 packages 來模擬條件編譯的想法。根據這一想法，他在 Java 裡頭產生原本在 C 語言中極爲有用的 *assertion* 機制。憑藉此一機制，你可以宣告「這應該是 true」或「這應該是 false」。一旦某個述句不符合你所宣告的真僞狀態，你便會發現它。這樣的工具在除錯過程中相當實用。

下面便是你可以用來協助除錯的 class：

```
//: com:bruceeckel:tools:debug:Assert.java
// Assertion tool for debugging.
package com.bruceeckel.tools.debug;
```

```
public class Assert {
  private static void perr(String msg) {
    System.err.println(msg);
  }
  public final static void is_true(boolean exp) {
    if(!exp) perr("Assertion failed");
  }
  public final static void is_false(boolean exp){
    if(exp) perr("Assertion failed");
  }
  public final static void
  is_true(boolean exp, String msg) {
    if(!exp) perr("Assertion failed: " + msg);
  }
  public final static void
  is_false(boolean exp, String msg) {
    if(exp) perr("Assertion failed: " + msg);
  }
} ///:~
```

這個 class 只是封裝了 Boolean 檢驗動作，並在檢驗失敗時列印錯誤訊息。你會在第 10 章學到更複雜的錯誤處理工具，該工具稱為「異常處理（*exception handling*）」。此刻，**perr()** 便足夠我們用的了。

上述 class 的輸出結果會因為「將訊息寫至 **System.err**」而列印於主控台（console）標準示誤串流（*standard error*）中。

如果你想使用這個 class，可以在程式中加入此行：

```
import com.bruceeckel.tools.debug.*;
```

如果你想於出貨時取消這個 assertions 功能，可以開發第二個 **Assert** class，並將它置於另一個不同的 package 中：

```
//: com:bruceeckel:tools:Assert.java
// Turning off the assertion output
// so you can ship the program.
package com.bruceeckel.tools;

public class Assert {
  public final static void is_true(boolean exp){}
```

```
  public final static void is_false(boolean exp){}
  public final static void
  is_true(boolean exp, String msg) {}
  public final static void
  is_false(boolean exp, String msg) {}
} ///:~
```

現在，如果你將先前的 **import** 述句改爲：

```
import com.bruceeckel.tools.*;
```

程式就不再印出 assertions 訊息了。以下即爲一例：

```
//: c05:TestAssert.java
// Demonstrating the assertion tool.
// Comment the following, and uncomment the
// subsequent line to change assertion behavior:
import com.bruceeckel.tools.debug.*;
// import com.bruceeckel.tools.*;

public class TestAssert {
  public static void main(String[] args) {
    Assert.is_true((2 + 2) == 5);
    Assert.is_false((1 + 1) == 2);
    Assert.is_true((2 + 2) == 5, "2 + 2 == 5");
    Assert.is_false((1 + 1) == 2, "1 +1 != 2");
  }
} ///:~
```

改而匯入不同的 **package**，你便可以將程式碼從除錯版改爲出貨版。這個技巧可用於任何類型的條件編譯程式碼上。

使用 package 的一些忠告

當你產生 package 時，在給定 package 名稱後，即隱隱指定了某個目錄結構。這個 package 必須置於其名稱所指的目錄之中。從 CLASSPATH 所含括的目錄出發，必須能夠搜尋至此目錄。開始學習運用關鍵字 **package** 時，結果可能令人有點沮喪。因爲除非你遵守「package 名稱對應至目錄路徑」的規則，否則會得到許多不可思議的執行期錯誤訊息，告訴你無法尋得某些特定 classes — 即使它們其實位於同一目錄中。如果你得到類似

訊息，請試著將 **package** 述句註解掉。如果這麼做便可執行的話，你就知道問題出在哪裡了。

Java 存取權限飾詞（access specifiers）

Java 存取權限飾詞 **public**、**protected**、**private** 應該置於 class 內的每個成員的定義式前，無論此成員究竟是資料成員或函式。每個飾詞僅控制它所修飾的那一份定義的存取權限。這和 C++的存取權限飾詞形成明顯對比。在 C++中，存取權限飾詞控制著緊接於其後的所有定義式，直到另一個存取權限飾詞出現。

每一樣東西都需要被指定某種存取權限。接下來各節中，你會學到各類存取權限。

"Friendly"（友善的）

本章之前的所有程式範例，都沒有給定任何存取權限飾詞，那麼會發生什麼事呢？雖然預設的存取權限不需要任何關鍵字，但通常被稱爲 friendly。意思是同一個 package 內的其他所有 classes 都可以存取 friendly 成員，但對 package 以外的 classes 則形同 **private**。由於一個編譯單元（檔案）僅能隸屬於一個 package，所以同一編譯單元內的所有 classes 都視彼此爲 friend。也因此 friendly 存取權限又稱爲 package 存取權限。

friendly 存取權限讓你得以將相關的 classes 置於同一個 package 中，使它們之間得以輕易進行互動。當你將 classes 置於某個 package（授予相互存取 friendly 成員的權力，也就是使它們成爲朋友），你等於「擁有」該 package 內的程式碼。「只有你自己的程式碼，才可以友善地存取你所擁有的其他程式碼」這種想法是合理的。可以說，在「將 classes 置於同一個 package」這件事情上面，friendly 存取權限給了很好的目的或理由。許多程式語言對於「如何將各檔案中的定義組織起來」的處理方式可能極爲混亂，但 Java 強迫你以合理的方式加以組織。此外，你或許會想排除那些

「不應存取目前 package 內的 classes」的一些 classes。

class 手握「讓誰誰誰具有存取我的成員的權限」的鑰匙。其他 classes 不能憑空得到存取權限。其他 package 的程式碼不能一現身就說：「嗨，我是 **Bob** 的朋友！」然後就希望看到 **Bob** 的 **protected**、friendly、**private** 成員。如果想要對外授予某個成員的存取權限，唯一的方法是：

1. 宣告該成員為 **public**。那麼任何人在任何地方皆得以存取之。
2. 不加任何存取權限飾詞，使該成員成為 friendly，並將其他 classes 置於同一個 package 內，於是其他 classes 便可存取該成員。
3. 就如第 6 章即將見到的，當我們引入繼承關係，繼承下來的 class 能夠存取 **protected** 成員（但不含括 **private** 成員），就和存取 **public** 成員一樣。只有當兩個 classes 位於同一個 package 時，才能存取對方的 friendly 成員。不過此刻毋需擔憂此事。
4. 提供「存取式（accessor）/變異式（mutator）」（也被稱為 "get/set" 函式），藉以取值和設值。就 OOP 來說，這是最進步的方式，也是 JavaBeans（第 13 章介紹）的根基所在。

public：介面存取（interface access）

當你使用關鍵字 **public**，代表「緊接於 **public** 之後的成員將可為每個人所用」，特別是對於此一程式庫的使用者（客端程式員）。假設你定義了名為 **dessert** 的 package，它含有以下編譯單元：

```
//: c05:dessert:Cookie.java
// Creates a library.
package c05.dessert;

public class Cookie {
  public Cookie()
   System.out.println("Cookie constructor");
  }
  void bite() { System.out.println("bite"); }
} ///:~
```

請務必記得，**Cookie.java** 必須置於 **c05**（代表本書第 5 章）目錄下的 **dessert** 子目錄中，而 **c05** 亦必須位於 CLASSPATH 所列的某一個目錄下。千萬別誤以為 Java 一定會將目前所在目錄做為搜尋起點之一。如果你沒有將 '.'（譯註：目前所在目錄）含括在你的 CLASSPATH 中，Java 便不會把目前所在目錄當做搜尋起點。

現在，如果你撰寫某個程式並在其中使用 **Cookie**：

```
//: c05:Dinner.java
// Uses the library.
import c05.dessert.*;

public class Dinner {
  public Dinner() {
   System.out.println("Dinner constructor");
  }
  public static void main(String[] args) {
    Cookie x = new Cookie();
    //! x.bite(); // Can't access
  }
} ///:~
```

你便可以產生 **Cookie** 物件，因為 **Cookie** 的建構式是 **public** 而其本身也是 **public**（稍後我們將學到更多關於 **public** class 的觀念）。不過 **Dinner.java** 之中無法取用 **bite()**，因為 **bite()** 的存取權限是 friendly，僅供 package **dessert** 內部取用。

default（預設的）package

你可能會很驚訝地發現，下面的程式碼竟然可以順利編譯，即便有些地方並不遵守規則：

```
//: c05:Cake.java
// Accesses a class in a
// separate compilation unit.

class Cake {
  public static void main(String[] args) {
    Pie x = new Pie();
    x.f();
  }
```

```
} ///:~
```

第二個檔案位於同一目錄中：

```
//: c05:Pie.java
// The other class.

class Pie {
  void f() { System.out.println("Pie.f()"); }
} ///:~
```

一開始你可能會將上述兩個檔案視爲完全無關的檔案，但 **Cake** 卻能夠產生 **Pie** 物件並呼叫其 **f()**！（請注意，如果想編譯這些檔案，你得將 '.' 加至你的 CLASSPATH 環境變數中）。你會很自然地認爲 **Pie** 和 **f()** 的存取權限是 friendly，因此不可被 **Cake** 所用。是的，它們的確是 friendly。**Cake.java** 之所以可以存取它們，原因是它們位於同一個目錄中，而且沒有爲自己設定任何 package 名稱。Java 會自動將這兩個檔案視爲隸屬於該目錄的所謂 default package 中，因此對同目錄下的其他檔案來說，它們都是 friendly。

private: 不要碰我！

關鍵字 **private** 表示「除了當事人（某個成員）所在的 class，沒有任何人可以存取這個成員」。即使是同一個 package 內的其他 classes，也無法存取你的 **private** 成員。這種宣告方式無異隔離自己。但是從另一個角度說，多人協力開發同一個 package 的情況並非不可能，因此，**private** 讓你得以自由更動你的成員，無需擔心這麼做是否影響同一個 package 下的其他 class。

預設的 friendly 存取權限（或稱 package 存取權限）通常已經足以提供堪用的隱藏性質；請記住，package 的使用者無法取用 friendly 成員。這是好事，因爲所謂預設的存取權限，應該是你在正常情況下使用的存取權限（也是忘了加上任何飾詞時會生效的存取權限）。因此通常你會特別思考的是「希望明確開放給客端程式員使用」的一些成員，並將它們宣告爲 **public**。一開始你可能不認爲你會時常用到 **private** 關鍵字，因爲少了它

還是可以的（這和 C++形成明顯對比）。不過事實證明，**private** 極爲重要，尤其在多緒環境下（multithreading，詳見第 14 章）。

下面是 **private** 的使用範例：

```
//: c05:IceCream.java
// Demonstrates "private" keyword.

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();
  }
} ///:~
```

這個例子說明了 **private** 終有其用：您可能會想控制物件的生成方式，並且不允許其他人直接取用某個特定建構式（或所有建構式）。上例中你無法透過 **Sundae** 的建構式來產生物件，你得呼叫 **makeASundae()**，讓它來爲你服務[2]。

只要你確信某些函式對 class 而言只扮演「後勤支援」性質，你都可以將它們宣告爲 **private**，以確保不會在同一個 package 的其他地方誤用到它們，而自己又能保有更動、甚至刪除的權力。將某個函式宣告爲 **private**，可保證你自己仍握有決定權。

對於 class 內的 **private** 資料成員而言，情形亦同。除非你想曝露底層實作細目（這是一種很難想像的罕見情境），否則您應該讓所有的資料成員都

---

[2] 此處還會產生其他效應。因爲我們僅定義 *default* 建構式，而它又是 **private**，所以這麼做同時也杜絕了以此 class 爲根源的繼承行爲（詳見第 6 章）。

成爲 **private**。不過，在 class 中令某個物件的 reference 爲 **private**，並不代表其他物件無法擁有該物件的 **public** reference。請參考附錄 A 中關於 aliasing（別名）的種種討論。

protected：幾分友善

想要了解 **protected** 存取權限，我們得先做點跳躍動作。首先你應該知道，本書開始介紹繼承機制（第 6 章）時，你才需要了解本節內容。但基於完整性考量，我還是在此處提供 **protected** 的簡要說明和使用範例。

關鍵字 **protected** 所處理的是所謂的「繼承（*inheritance*）」觀念。在此觀念中，我們可以將新的成員加到被我們稱爲 base class（基礎類別）的既有 class 中，而無需碰觸 base class。你也可以改變 base class 既有成員的行爲。如果想繼承某個既有的 class，你可以讓新 class 延伸（**extends**）既有的 class，就像這樣：

```
class Foo extends Bar {
```

至於 class 定義式中的其他部份看起來相同。

如果你產生新的 package，並且繼承另一個 package 中的 class，那麼你就只能存取原先 package 中的 **public** 成員（當然啦，如果你是在同一個 package 中施行繼承動作，你將仍舊擁有對所有 friendly 成員的一般性「package 存取權限」。有時候，base class 開發者會希望允許其 derived classes（衍生類別）存取某個特定成員，但不希望所有 classes 都有此權力。這正是 **protected** 的用途。以先前出現的那個 **Cookie.java** 爲例，以下 class 將無法存取其 friendly 成員：

```
//: c05:ChocolateChip.java
// Can't access friendly member
// in another class.
import c05.dessert.*;

public class ChocolateChip extends Cookie {
  public ChocolateChip() {
   System.out.println(
     "ChocolateChip constructor");
```

```
  }
  public static void main(String[] args) {
    ChocolateChip x = new ChocolateChip();
    //! x.bite(); // Can't access bite
  }
} ///:~
```

關於繼承，最有趣的事情之一便是，如果 class **Cookie** 擁有 **bite()**，那麼後者也會存在於繼承自 **Cookie** 的所有 classes 中。但由於 **bite()** 是另一個 package 中的 friendly 函式，所以無法被這些 derived classes 取用。你當然可以將它宣告為 **public**，但這麼一來每個人都可以取用，這可能不是你想要的。如果我們將 class **Cookie** 改成這樣子：

```
public class Cookie {
  public Cookie()
    System.out.println("Cookie constructor");
  }
  protected void bite() {
    System.out.println("bite");
  }
}
```

那麼，在 package **dessert** 中，**bite()** 仍舊具備 friendly 權限，而任何繼承自 **Cookie** 的 classes 亦可加以取用。這和 **public** 並不相同。

Interface（介面）與 implementation（實作）

存取權限的控制通常被視為是一種「實作細目的隱藏（*implementation hiding*）」。將 class 內的資料和處理資料的行為包裝起來，結合實作細目之隱藏，即是所謂的封裝（*encapsulation*）[3]。其結果就是一個兼具特徵（characteristics。譯註：此處意指資料）和行為的資料型別。

[3] 即是單單只是實作細目的隱藏，人們也常稱之為封裝。

基於兩個理由，我們需要控制存取權限，在資料型別中建立諸般界限。第一，建立起一道界限，指明哪些是客端程式員可使用的，哪些是他們不可使用的。於是你便可以將內部機制建於結構之中，無需擔心客端程式員不小心將僅供內部使用的部份當做他們可使用的一部份介面。

上述原因直接影響了第二個理由 – 將介面和實作分離。如果某個結構被用於一組程式中，而客端程式員除了發送訊息給 **public** 介面（譯註：亦即呼叫 **public** 函式），完全無法進行任何動作，那麼你便可以在不修改客端程式碼的情況下更動所有 non-**public**（包括 friendly、**protected**、**private**）成員。

我們正置身於物件導向編程世界中。在這個世界裡，**class** 實際上所描述的是「某一類物件」，就像你描述某一種魚或某一種鳥一樣。任何隸屬於某個 class 的所有物件都具備同樣的特徵（譯註：資料）和行爲。所謂 class，就是所有同型物件的外觀長相及行爲舉措的描述。

在最原始的物件導向程式語言 Simula-67 中，關鍵字 **class** 被用來描述新的資料型別。這個關鍵字亦被沿用於大多數物件導向程式語言。這種語言所關注的焦點是：產生一些「不僅只是內含資料和函式」的新資料型別。

Java 中的 class 是極爲基礎的 OOP 觀念。它是本書不以粗體表示的關鍵字之一，因爲對於出現如此頻繁的字來說，以粗體表示實在太煩人了。

爲了讓程式更清楚，你可能會喜歡將 **public** 成員置於 class 起始處，其後再接著 **protected** 成員、friendly 成員、**private** 成員。這種作法的優點是，class 使用者可以由上而下，先看到對他們來說最爲重要的部份（亦即 **public** 成員，因爲它們可於檔案之外被取用），而在看到 non-**public** 成員時停下來，因爲已經抵達內部實作細節：

```
public class X {
  public void pub1( ) { /* . . . */ }
  public void pub2( ) { /* . . . */ }
```

```
  public void pub3( ) { /* . . . */ }
  private void priv1( ) { /* . . . */ }
  private void priv2( ) { /* . . . */ }
  private void priv3( ) { /* . . . */ }
  private int i;
  // . . .
}
```

但是這種寫法只能稍微增加程式的易讀性，因爲介面和實作仍舊混在一起。也就是說你仍然會看到原始碼（所謂實作部份），因爲它們就在 class 之內。此外 **javadoc** 所提供的「寓文件於註解」的功能（第 2 章介紹過）也降低了程式碼可讀性對客端程式員的重要性。將 class 介面呈現給其使用者的責任，其實應該由 *class browser*（類別瀏覽器）擔負起來。這是一種用以檢閱所有可用之 classes，並顯示在它們身上能夠進行什麼動作（亦即顯示出可用成員）的一種工具。當你閱讀本書時，這類瀏覽器應該已經成爲優良的 Java 開發工具中不可或缺的標準配備了吧。

Class 的存取權限

Java 的存取權限飾詞也可以用來決定「程式庫中哪些 classes 可以被程式庫使用者所用」。如果你希望某個 class 可被客端程式員所用，你得將關鍵字 **public** 置於 class 主體之左大括號前某處。爲 classes 而設的存取權限，可以控制客端程式員是否有權力產生某個 class 的物件。

如果你想控制 class 的存取權限，飾詞必須置於關鍵字 **class** 之前。因此，你可以這麼寫：

```
public class Widget {
```

現在，如果你的程式庫名爲 **mylib**，所有客端程式員都可以經由以下這種寫法來取用 **Widget**：

```
import mylib.Widget;
```

或

```
import mylib.*;
```

不過，這裡還是存在一些額外限制：

1. 每個編譯單元（檔案）都僅能有一個 **public** class。其中的觀念是，每個編譯單元都擁有一個由 public class 所表現的單一 **public** 介面。當然，編譯單元內可以存在許多個支援用的 friendly classes。如果編譯單元中的 **public** class 不只一個，編譯器會給你錯誤訊息。

2. **public** class 的名稱（含大小寫）必須恰與其編譯單元（檔案）名稱相符。所以，對 **Widget** 而言，其檔名必須是 **Widget.java** 而不能是 **widget.java** 或 **WIDGET.java**。如果不是這樣，會出現編譯錯誤。

3. 雖然通常不會這麼做，但編譯單元內的確可以不含任何 **public** class。這種情況下，你可以任意給定檔案名稱。

假設你在 **mylib** 中撰寫某個 class，僅僅只是爲了用它來協助完成 **Widget** 或 **mylib** 內的其他 **public** class 的工作。你不想爲了撰寫說明文件給客端程式員看而傷腦筋，而且你認爲一段時間之後，你可能會徹底改變原有作法並完全捨棄舊版本，以全新版本替代。如果想擁有上述彈性，你得確保沒有任何客端程式員倚賴 **mylib** 內的特定實作細目。欲達此一目的，只要拿掉 class 的 **public** 飾詞，它就成了 friendly（那麼它也就只能用於 package 內部了）。

請注意，class 不能是 **private**（這麼做會使得除了它自己沒有任何 class 可加以取用）或 **protected**[4]。所以，對於 class 存取權限，你只有兩個選擇：friendly 或 **public**。如果你不希望其他任何人取用某個 class，請將其所有建構式宣告爲 **private**，這麼一來便可阻止任何人產生其物件，唯有

[4] 事實上 *inner class*（內隱類別）可以是 **private** 或 **protected**，不過這是特例。詳見第 7 章。

一個例外，那就是在 class **static** 成員中可以辦到[5]。下面便是一例：

```
//: c05:Lunch.java
// Demonstrates class access specifiers.
// Make a class effectively private
// with private constructors:

class Soup {
  private Soup() {}
  // (1) Allow creation via static method:
  public static Soup makeSoup() {
    return new Soup();
  }
  // (2) Create a static object and
  // return a reference upon request.
  // (The "Singleton" pattern):
  private static Soup ps1 = new Soup();
  public static Soup access() {
    return ps1;
  }
  public void f() {}
}

class Sandwich { // Uses Lunch
  void f() { new Lunch(); }
}

// Only one public class allowed per file:
public class Lunch {
  void test() {
    // Can't do this! Private constructor:
    //! Soup priv1 = new Soup();
    Soup priv2 = Soup.makeSoup();
    Sandwich f1 = new Sandwich();
    Soup.access().f();
  }
} ///:~
```

[5] 你也可以透過繼承（第 6 章）辦到。

到目前爲止，我們所用的大多數函式皆傳回 **void** 或基本型別，因此以下定義乍見之下令人有點困惑：

```
public static Soup access() {
  return ps1;
}
```

上述函式名稱（**access**）之前的字，指出該函式回傳的東西。截至目前，在我們的例子中，此字常常是 **void**，表示不回傳任何東西。但你也可以回傳一個 object reference，這就是上述式子的作爲。上面這個函式所回傳的，即是一個 reference，代表一個 class **Soup** 物件。

class Soup 示範如何將所有建構式都宣告爲 **private** 以防止直接產生某個 class 物件。請千萬記住，如果你沒有自行撰寫至少一個建構式，會有一個 *default* 建構式（不具任何引數的建構式）被自動合成出來。如果我們撰寫自己的 *default* 建構式，它就不會被自動合成；如果我們讓它成爲 **private**，就沒有任何人能夠產生這個 class 的物件。但是這麼一來別人又該如何使用這個 class 呢？上述例子示範了兩種作法，一是撰寫 **static** 函式來產生新的 **Soup** 並回傳其 reference。如果你希望在執行 **Soup** 之前先進行某些額外處理，或希望記錄（或限制）究竟有多少個 **Soup** 物件被產生出來，這種作法十分有用。

第二種作法是使用某個設計樣式（*design patterns*，這個主題涵蓋於《*Thinking in Patterns with Java*》書中，可於 *www.BruceEckel.com* 下載）。這個樣式被稱爲 "singleton"，因爲它讓整個程式面對某個 class 時只能產出唯一一個物件。class **Soup** 的物件是被 **Soup** 的 **static private** 成員函式產生出來的，所以恰恰只能有一份。而且，除了透過 **public access()** 加以存取，別無它法。

如前所述，如果你未指定某個 class 的存取權限，預設便是 friendly。這表示同一個 package 內的其他 classes 能夠生成該 class 的物件，而 package 之外則否。（請千萬記得，同一目錄中的所有檔案，如果沒有明確的 **package** 宣告，都會被視爲是該目錄的 *default* package。）不過，如果該 class 有某個 **static public** 成員，那麼客端程式員即使無法生成該 class 的物件，仍然能夠存取這個 **static** 成員。

摘要

在任何相互關係中，讓多個參予者共同遵循某些界限，是相當重要的事。當你開發程式庫時，會建立起和使用者（亦即客端程式員，也就是將你的程式庫結合至應用程式，或藉以開發更大型程式庫的人）之間的關係。

如果缺乏規範，客端程式員可以對 class 的成員進行他們想進行的任何動作 － 即使你可能不希望他們直接操作這些成員。喔，每樣東西都攤在全世界面前。

本章討論如何將眾多的 classes 組成程式庫。首先介紹如何將一組 classes 包裝於程式庫中，然後介紹如何控制 class 成員的存取權限。

據估計，以 C 語言來開發專案，大概發展五萬行至十萬行程式碼時，就會開始出現問題。因為 C 僅有單一命名空間，所以容易發生命名衝突問題，引發許多額外的管理代價。Java 語言的 **package** 關鍵字、package 命名架構、**import** 關鍵字，讓你得以完全掌控命名機制，所以命名衝突的問題可輕易規避之。

我們之所以要控制成員的存取權限，基於兩個理由。首先，讓使用者無法碰觸他們不該碰觸的東西；這些東西僅供資料型別內部機制所用，不在「使用者賴以解決問題」的介面之中。因此，將這些函式和資料成員宣告為 **private**，對使用者來說是一種服務。使用者可以因此輕易看出，哪些東西對他們來說是重要的，哪些東西對他們來說可以略而不見。如此一來便可以減輕他們「認識 class」的負擔。

存取權限控制的第二個存在理由，同時也是最重要的理由，就是讓程式庫設計者可以更動 class 內部運作方式，而無需擔心波及客端程式員。一開始你可能會以某種方式開發 class，然後發現如果更改程式結構，可以提高執行速度。一旦介面和實作可以被明確地加以隔離和保護，此一目標便可達成，無需強迫程式庫使用者重新改寫程式碼。

Java 的「存取權限飾詞」賦予 classes 開發者極具價值的控制能力。classes 使用者可以清楚看出，哪些是他們可以使用的，哪些是他們應該略而不見的。更重要的是，這能夠確保不存在任何使用者「倚賴 class 底層實作細節」。身為 classes 開發者，如果你的任何改變可以完全不干擾你的使用者，你便可以安心改變你的底層實作，因為客端程式員無法存取 class 的這一部份。

當你擁有改變底層實作的能力，你不僅可以在日後改善你的設計，也擁有了犯錯的自由。因為不論多麼小心翼翼地規劃和設計，人終究難免犯錯。當你知道犯下錯誤但相對安全的時候，你便能夠更放心地進行實驗，以更快的速度學習，並更早完成專案。

class 的公開（public）介面，是使用者看得到的部份。因此在分析與設計階段，這一部份也是 class 正確與否的關鍵。即使如此，你仍對它擁有些許改變和調整的空間。如果你沒有一開始便製作出正確的介面，你可以於日後加入額外的函式 — 是的，只要不刪去客端程式員已經用於應用程式中的任何東西，都行。

練習

某些經過挑選的題目，其解答置於《*The Thinking in Java Annotated Solution Guide*》電子文件中。僅需小額費用便可自 *www.BruceEckel.com*.網站取得。

1. 撰寫一個程式，在其中產生 **ArrayList** 物件，但不明確匯入 **java.util.***。
2. 將標題為「package：程式庫單元（library unit）」一節中與 **mypackage** 有關的程式碼片段，改寫為一組可編譯、可執行的 Java 檔案。
3. 將標題為「衝突」一節中的程式碼片段，改寫為完整程式，並檢查實際發生的命名衝突。
4. 將本章所定義的 **P** class 更一般化：加入 **rint()** 和 **rintln()** 的所有重載版本，使其足以處理所有 Java 基本型別。

5. 改變 **TestAssert.java** 中的 **import** 述句，試著開啓、關閉 assertion 機制。

6. 撰寫一個具有 **public**、**private**、**protected**、friendly 等等資料成員和成員函式的 class。爲它產生一個物件並進行觀察：當你嘗試取用所有 class 成員時，會產生什麼類型的編譯訊息？注意，位於同一目錄中的所有 classes，都被設定於 *default* package 內。

7. 撰寫一個 class，令它具備 **protected** 資料。並在同一個檔案中撰寫第二個 class，爲此 class 提供函式，使它操作第一個 class 的 **protected** 資料。

8. 改寫標題爲「protected：幾分友善」一節中的 **Cookie**。驗證其中的 **bite()** 並非 **public**。

9. 你可以在標題爲「Class 的存取權限」一節中，找到描述 **mylib** 和 **Widget** 的程式片段。請完成這個程式庫，並撰寫一個不在 **mylib** package 中的 **Widget** class。

10. 建立一個新目錄，並將它加到你的 CLASSPATH 環境變數中。將 **P.class** 檔（**com.bruceeckel.tools.P.java** 編譯後的產品）複製到新目錄，並改變檔案名稱、內部的 **P** class 名稱、及其函式名稱。你可能會想加入其他輸出訊息，藉以觀看運作方式。在另一個目錄下撰寫這個新的 class 的應用程式。

11. 請你依循範例程式 **Lunch.java** 中的格式，撰寫一個名爲 **ConnectionManager** 的 class，使其能夠管理固定大小的 array 中的 **Connection** 物件。請你限制客端程式員，使他們無法自行產生 **Connection** 物件，只能經由 **ConnectionManager** 的 **static** 函式來獲得這些物件。當 **ConnectionManager** 之中不再存有任何物件時，便回傳 **null** reference。請在 **main()** 中測試這兩個 classes。

12. 在 c05/local 目錄下（這應該記錄於你的 CLASSPATH 環境變數），建立如下檔案：

```
///: c05:local:PackagedClass.java
package c05.local;
```

```
class PackagedClass {
  public PackagedClass() {
    System.out.println(
      "Creating a packaged class");
  }
} ///:~
```

然後在 c05 之外的另一目錄中產生如下檔案：

```
///: c05:foreign:Foreign.java
package c05.foreign;
import c05.local.*;
public class Foreign {
   public static void main (String[] args) {
      PackagedClass pc = new PackagedClass();
   }
} ///:~
```

請解釋爲什麼編譯器會發出錯誤訊息。如果將 **Foreign** class 置於 **c05.local** package 內，能夠改變什麼嗎？

6: 重複運用 Classes

Java 有著眾多令人讚嘆的功能，程式碼的重複運用便是其中之一。但是，如果想獲得革命性的改變，你得遠遠超越「複製程式碼、然後改變之」的舊有模式。

C 之類的程序性（procedural）語言便採用這種舊方法，但是沒有得到很好的效果。就像 Java 中的所有事物一樣，解決之道圍繞在 class 身上。你可以產生新的 classes 來重複運用程式碼，不須重頭寫起。你可以使用某人已經開發好、除錯完畢的既有 classes。

此中秘訣便在於能夠使用既有的 classes 而不破壞其程式碼。你會在本章看到，兩種方法足以達成上述目的。第一種方法十分直觀：在新的 class 中產生既有 class 的物件。這種方法稱爲「複合（*composition*）」，或稱組合，因爲新的 class 是由既有 classes 的物件組成。這種情形只是很單純地重複運用既有程式碼的功能，而非重複運用其形式。

第二種方法更爲精巧，能夠讓新的 class 成爲既有 class 的一類。你可以實際接收既有 class 的形式，並加入新碼，無需更動既有的 class。這種神奇行爲被稱爲「繼承（*inheritance*）」，而且編譯器能夠爲你完成大部份工作。繼承是物件導向程式設計的基石之一，第 7 章還會探究其深遠意涵。

對複合（*composition*）和繼承（*inheritance*）而言，語法及行爲大多類似（這麼做饒富意義，因爲二者都是從既有型別產生出新型別）。你可以在本章之中學到這兩種「程式碼重複運用」的機制。

複合（Composition）語法

本書至此，屢屢使用複合技術。只要將 object references 置於新的 classes 中即是。假設你想擁有某個物件，它必須能夠儲存多個 **String** 物件、兩三

個基本型別資料、以及另一個 class 物件。你可以直接定義基本型別資料，但是對非基本型別的物件來說，你得將其 references 置於新的 class 內：

```
//: c06:SprinklerSystem.java
// Composition for code reuse.

class WaterSource {
  private String s;
  WaterSource() {
    System.out.println("WaterSource()");
    s = new String("Constructed");
  }
  public String toString() { return s; }
}

public class SprinklerSystem {
  private String valve1, valve2, valve3, valve4;
  WaterSource source;
  int i;
  float f;
  void print() {
    System.out.println("valve1 = " + valve1);
    System.out.println("valve2 = " + valve2);
    System.out.println("valve3 = " + valve3);
    System.out.println("valve4 = " + valve4);
    System.out.println("i = " + i);
    System.out.println("f = " + f);
    System.out.println("source = " + source);
  }
  public static void main(String[] args) {
    SprinklerSystem x = new SprinklerSystem();
    x.print();
  }
} ///:~
```

WaterSource 所定義的函式中，有一個很不同尋常：**toString()**。稍後你會學到，每個非基本型別的物件都具備 **toString()**，當編譯器希望得到一個 **String**，而你手上卻只有那些物件的情況下，這個函式便會被喚起。所以下列算式：

```
System.out.println("source = " + source);
```

會讓編譯器知道你企圖將 **String** 物件（**"source = "**）和 **WaterSource** 相加。這對編譯器來說不具意義，因爲你只能將 **String** 加至另一個 **String**。所以編譯器說話了：『我將呼叫 **toString()**，把 **source** 轉爲一個 **String**！』完成這個動作後它便能夠將兩個 **String** 合併在一塊兒，並將結果傳給 **System.out.println()**。如果你希望你的 class 具備這種行爲，只要爲它撰寫 **toString()** 即可。

乍見之下你可能會假設編譯器自動爲上述程式碼中的每一個 references 產生相應的物件（喔是的，Java 應該表現出安全謹慎的一面）；例如它會呼叫 **WaterSource** 的 *default* 建構式，將 **source** 初始化。但印出來的結果卻是：

```
valve1 = null
valve2 = null
valve3 = null
valve4 = null
i = 0
f = 0.0
source = null
```

是的，class 之中屬於基本型別的資料欄位，的確會被自動初始化爲零，就如我在第 2 章所言。但 object references 會被初始爲 **null**，而且如果你試著透過這些 reference 呼叫任何函式，會引發異常（exception）。如果可以印出其內容而不出現異常，對我們來說是好事（而且也很實用）。

編譯器「不爲每個 reference 產生預設物件」是有意義的，因爲這麼做在許多情況下造成不必要的負擔。如果你希望初始化這些 references，你可以在下列地點進行：

1. 在物件定義處。這表示它們一定能夠在「建構式被呼叫前」完成初始化動作。
2. 在 class 建構式中。
3. 在你實際需要用到該物件的地方。這種方式常被稱爲「緩式初始化（*lazy initialization*）」。在無需每次都產生物件的場合中，這種作法可以降低額外負擔。

以下例子同時示範了上述三種作法：

```
//: c06:Bath.java
// Constructor initialization with composition.

class Soap {
  private String s;
  Soap() {
    System.out.println("Soap()");
    s = new String("Constructed");
  }
  public String toString() { return s; }
}

public class Bath {
  private String
    // Initializing at point of definition:
    s1 = new String("Happy"),
    s2 = "Happy",
    s3, s4;
  Soap castille;
  int i;
  float toy;
  Bath() {
    System.out.println("Inside Bath()");
    s3 = new String("Joy");
    i = 47;
    toy = 3.14f;
    castille = new Soap();
  }
  void print() {
    // Delayed initialization:
    if(s4 == null)
      s4 = new String("Joy");
    System.out.println("s1 = " + s1);
    System.out.println("s2 = " + s2);
    System.out.println("s3 = " + s3);
    System.out.println("s4 = " + s4);
    System.out.println("i = " + i);
    System.out.println("toy = " + toy);
    System.out.println("castille = " + castille);
```

```
  }
  public static void main(String[] args) {
    Bath b = new Bath();
    b.print();
  }
} ///:~
```

注意，**Bath** 建構式中，有一行述句在任何初始化動作之前執行。如果你未在定義處進行初始化，那麼就無法保證在你發送訊息給 object references 之前能夠進行任何初始化 — 反倒是無可避免會發生執行期異常。

以下是這個程式的輸出：

```
Inside Bath()
Soap()
s1 = Happy
s2 = Happy
s3 = Joy
s4 = Joy
i = 47
toy = 3.14
castille = Constructed
```

當 **print()** 被呼叫，它會填寫 **s4** 的值。因此所有資料欄位（fields）在它們被使用之際都已被妥善初始化了。

繼承（Inheritance）語法

繼承是 Java（一般而言也是 OOP 語言）不可或缺的一個部份。事實上當你撰寫 class 的同時便已進行了繼承。是的，即使沒有明確指出要繼承某個 class，你仍然會繼承 Java 的標準根源類別 **Object**。

「複合」語法平淡無奇，「繼承」則必須以截然不同的形式爲之。當你進行繼承，你得宣告「新 class 和舊 class 是類似的」。和平常一樣，首先給定 class 名稱，但是在寫下 class 主體左大括號前，先寫下關鍵字 **extends**，其後緊接著 base class（基礎類別）名稱，這樣便完成了繼承宣告動作。至此便自動獲得了 base class 的所有資料成員和成員函式。

以下便是一例：

```
//: c06:Detergent.java
// Inheritance syntax & properties.

class Cleanser {
  private String s = new String("Cleanser");
  public void append(String a) { s += a; }
  public void dilute() { append(" dilute()"); }
  public void apply() { append(" apply()"); }
  public void scrub() { append(" scrub()"); }
  public void print() { System.out.println(s); }
  public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    x.print();
  }
}

public class Detergent extends Cleanser {
  // Change a method:
  public void scrub() {
    append(" Detergent.scrub()");
    super.scrub(); // Call base-class version
  }
  // Add methods to the interface:
  public void foam() { append(" foam()"); }
  // Test the new class:
  public static void main(String[] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    x.print();
    System.out.println("Testing base class:");
    Cleanser.main(args);
  }
} ///:~
```

這個程式示範了許多特性。首先，在 **Cleanser append()** 中我以 += 運算子將 **String** 物件接續於 **s** 之後。+= 運算子是 Java 設計者重載後用以處理 **String**s 的運算子之一（另一個是 "+"）。

第二，**Cleanser** 和 **Detergent** 都含有 **main()**。你可以爲每個 classes 都撰寫 **main()**。通常我會建議你以這種方式撰寫程式碼，以便將測試碼包裝於 class 內。不過如果某個程式擁有許多 classes，僅有命令列所調用的那個 class 的 **main()** 會被喚起 — 這個 **main()** 必須是 **public**，其所屬 class 則無所謂是否爲 **public**。在這個例子中，你的命令列可以是 **java Detergent**，於是 **Detergent.main()** 便被喚起。你也可以鍵入 **java Cleanser**，於是 **Cleanser.main()** 便被喚起 – 即使 **Cleanser** 並非 **public** class。這種「爲每個 class 提供 **main()**」的技巧，可以使每個 class 的單元測試（unit testing）更爲容易。而且在完成單元測試之後，無需刪去 **main()**；你可以將它留下以待日後再加測試。

在這裡你可以看到，**Detergent.main()** 明確呼叫 **Cleanser.main()**，並將命令列引數原封不動傳過去（當然你也可以傳入任意的 **String** array）。

這裡有一點很重要：**Cleanser** 的所有函式都是 **public**。請千萬記得，如果你未指定成員的存取權限，預設便是 friendly，也就是說只能在同一個 package 中加以取用。因此，如果沒有指定存取權限，同一個 package 中的所有 classes 皆能使用這些函式。這對 **Detergent** 沒有問題。但是如果位於另一個 package 中的某個 class 繼承了 **Cleanser**，它便僅能存取其 **public** 成員。所以，爲了繼承著想，一般原則是將所有資料成員宣告爲 **private**，將所有函式宣告爲 **public**（**protected** 成員也允許讓衍生的 classes 取用，這點稍後馬上會說明）。當然，你得針對特別情況做一些調整，但上述所言的確是個實用準則。

請注意，**Cleanser** 的介面含括了一組函式：**append()**、**dilute()**、**apply()**、**scrub()**、**print()**。由於 **Detergent** 衍生自 **Cleanser**（透過關鍵字 **extends**），所以它會自動從 **Cleanser** 的介面中取得所有函式。你可以將繼承視爲「介面的重複運用」。實作細目亦隨之繼承而來，但並非最主要目的。

正如 **scrub()** 所顯示，你可以修改定義於 base class 中的函式。在這個例子中，你可能會想要在新版本中呼叫 base class 函式。但在 **scrub()** 中你無法僅僅呼叫 **scrub()** 來達成目的，因爲這麼做會產生遞迴 — 這不是你要的。爲了解決這個問題，Java 提供了關鍵字 **super**，透過它便可以取用目前 class 所繼承的「*superclass*（超類別，父類別）」。因此，算式 **super.scrub()** 會呼叫 base class 內的 **scrub()**。

實施繼承時，你不一定非得使用 base class 函式不可。你也可以將新的函式加至 derived class，就像將函式加入一般 class 沒有兩樣：只要加以定義即可。**foam()** 便是一個例子。

在 **Detergent.main()** 中你可以看到，除了 **Detergent** 函式（例如 **foam()**）之外，你也可以呼叫 **Cleanser** 的所有可用的函式。

base class 的初始化

由於現在有兩個 classes 牽扯進來（base class 和 derived class），不單只是一個，所以當你想像 derived class 所產生的物件時，可能會感到難以理解。外界看來，它像是一個具有「和 base class 相同介面」的新 class，也許還多了些額外的函式和資料成員。但繼承不單只是複製 base class 的介面而已。當你產生 derived class 物件時，其中會包含 base class 子物件（*subobject*）。這個子物件就和你另外產生的 base class 物件一模一樣。外界看來，base class 子物件被包裝於 derived class 物件之內。

當然，將 base class 子物件正確地加以初始化，極爲重要。只有一種方法可以保證此事：呼叫 base class 建構式，藉以執行建構式中的初始化動作。base class 建構式具備了執行 base class 初始化動作的所有知識和權力。Java 編譯器會自動在 derived class 建構式中插入對 base class 建構式的呼叫動作。以下範例說明此一特性在三層繼承關係上的運作方式：

```
//: c06:Cartoon.java
// Constructor calls during inheritance.

class Art {
  Art() {
    System.out.println("Art constructor");
  }
}

class Drawing extends Art {
  Drawing() {
    System.out.println("Drawing constructor");
  }
}

public class Cartoon extends Drawing {
  Cartoon() {
    System.out.println("Cartoon constructor");
  }
  public static void main(String[] args) {
    Cartoon x = new Cartoon();
  }
} ///:~
```

這個程式的輸出，說明了前述的「建構式自動被喚起」行爲：

```
Art constructor
Drawing constructor
Cartoon constructor
```

你可以看到，建構動作會由 base class「往外」擴散。所以 base class 會在 derived class 建構式取用它之前，先完成本身的初始化動作。

即使你並未撰寫 **Cartoon()** 建構式，編譯器也會爲你合成一個 *default* 建構式，並在其中呼叫 base class 的建構式。

帶有引數（arguments）的建構式

上例各個 classes 都具備 *default* 建構式，也就是說它們都不帶有引數。對編譯器而言，呼叫它們很容易，因爲不需擔心應該傳入什麼引數。但如果你的 class 不具備 *default* 建構式，或如果你想呼叫帶有引數的 base class 建構式，你便得運用關鍵字 **super**，並搭配適當的引數列，明白寫作出呼叫動作：

```
//: c06:Chess.java
// Inheritance, constructors and arguments.

class Game {
  Game(int i) {
    System.out.println("Game constructor");
  }
}

class BoardGame extends Game {
  BoardGame(int i) {
    super(i);
    System.out.println("BoardGame constructor");
  }
}

public class Chess extends BoardGame {
  Chess() {
    super(11);
    System.out.println("Chess constructor");
  }
  public static void main(String[] args) {
    Chess x = new Chess();
  }
} ///:~
```

如果你未在 **BoardGame()** 中呼叫 base class 建構式，程式將無法順利編譯，因爲編譯器無法找到符合 **Game()** 形式的建構式。此外，對 base class 建構式的呼叫，必須是 derived class 建構式所做的第一件事（如果你做錯了，編譯器會提醒你）。

捕捉base 建構式的異常

上面說了，編譯器會強迫你將 base class 建構式呼叫動作置於 derived class 建構式起始處。這表示沒有任何動作可以發生在它之前。第 10 章會提到，這個限制也使得 derived class 建構式無法捕捉所有來自 base class 的異常。有時候這挺不方便的。

搭配複合（composition）及繼承（inheritance）

同時使用複合與繼承，是極爲常見的事。以下例子同時使用兩種技術，配合必要的建構式初始化，爲你示範更爲複雜的 class 撰寫手法。

```
//: c06:PlaceSetting.java
// Combining composition & inheritance.

class Plate {
  Plate(int i) {
    System.out.println("Plate constructor");
  }
}

class DinnerPlate extends Plate {
  DinnerPlate(int i) {
    super(i);
    System.out.println(
      "DinnerPlate constructor");
  }
}

class Utensil {
  Utensil(int i) {
    System.out.println("Utensil constructor");
  }
}
```

```java
class Spoon extends Utensil {
  Spoon(int i) {
    super(i);
    System.out.println("Spoon constructor");
  }
}

class Fork extends Utensil {
  Fork(int i) {
    super(i);
    System.out.println("Fork constructor");
  }
}

class Knife extends Utensil {
  Knife(int i) {
    super(i);
    System.out.println("Knife constructor");
  }
}

// A cultural way of doing something:
class Custom {
  Custom(int i) {
    System.out.println("Custom constructor");
  }
}

public class PlaceSetting extends Custom {
  Spoon sp;
  Fork frk;
  Knife kn;
  DinnerPlate pl;
  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);
    System.out.println(
      "PlaceSetting constructor");
```

```
  }
  public static void main(String[] args) {
    PlaceSetting x = new PlaceSetting(9);
  }
} ///:~
```

雖然編譯器會強迫你初始化 base classes，並且規定你一定得在建構式起始處完成，但它並不會看管你是否將成員物件也初始化，你得自己記得。

保證適當清理

Guaranteeing proper cleanup

Java 並不具備 C++的解構式（*destructor*）概念。所謂解構式是物件被摧毀時自動被呼叫的一個函式。這或許是因爲 Java 裡頭的「物件摧毀方式」很簡單，只要忘掉物件、讓垃圾回收器在必要時候回收其所佔記憶體，就可以了，你無需明確地加以摧毀。

多數時候這是好事，但有時候你的 class 可能會在其生命期中執行某些需要事後清理的動作。然而第 4 章告訴我們，你無法得知垃圾回收器被喚起的時機，也無法知道它是否會被喚起。所以，如果你希望清除 class 所留下的某些東西，你得自行撰寫特殊的函式來進行此事，並讓客端程式員確知他們得呼叫此一函式來完成清理工作。首要之務一如第 10 章所說，便是將此類清理動作置於 **finally** 子句中，以防異常發生。下面這個例子，是個能夠在螢幕上繪出圖案的電腦輔助設計系統：

```
//: c06:CADSystem.java
// Ensuring proper cleanup.
import java.util.*;

class Shape {
  Shape(int i) {
    System.out.println("Shape constructor");
  }
  void cleanup() {
    System.out.println("Shape cleanup");
```

```
  }
}

class Circle extends Shape {
  Circle(int i) {
    super(i);
    System.out.println("Drawing a Circle");
  }
  void cleanup() {
    System.out.println("Erasing a Circle");
    super.cleanup();
  }
}

class Triangle extends Shape {
  Triangle(int i) {
    super(i);
    System.out.println("Drawing a Triangle");
  }
  void cleanup() {
    System.out.println("Erasing a Triangle");
    super.cleanup();
  }
}

class Line extends Shape {
  private int start, end;
  Line(int start, int end) {
    super(start);
    this.start = start;
    this.end = end;
    System.out.println("Drawing a Line: " +
           start + ", " + end);
  }
  void cleanup() {
    System.out.println("Erasing a Line: " +
           start + ", " + end);
    super.cleanup();
  }
}
```

```
public class CADSystem extends Shape {
  private Circle c;
  private Triangle t;
  private Line[] lines = new Line[10];
  CADSystem(int i) {
    super(i + 1);
    for(int j = 0; j < 10; j++)
      lines[j] = new Line(j, j*j);
    c = new Circle(1);
    t = new Triangle(1);
    System.out.println("Combined constructor");
  }
  void cleanup() {
    System.out.println("CADSystem.cleanup()");
    // The order of cleanup is the reverse
    // of the order of initialization
    t.cleanup();
    c.cleanup();
    for(int i = lines.length - 1; i >= 0; i--)
      lines[i].cleanup();
    super.cleanup();
  }
  public static void main(String[] args) {
    CADSystem x = new CADSystem(47);
    try {
      // Code and exception handling...
    } finally {
      x.cleanup();
    }
  }
} ///:~
```

在此系統之中，每樣東西都是某種 **Shape**（而 **Shape** 本身又是某種 **Object**，因爲它暗自繼承了那個根源類別）。每個 class 除了使用 **super** 來呼叫 base class 的 **cleanup()** 之外，還會重新定義這個函式。這些特殊的 **Shape** classes（**Circle**、**Triangle**、**Line**）全都有個建構式進行「繪製」動作 — 雖然物件生命期中呼叫的每個函式，其實都可能做些需要清理的事。每個 class 都有專屬的 **cleanup()**，用以將不存於記憶體中的東西，回復至物件存在前的狀態。

main() 之中出現兩個前所未見的關鍵字：**try** 和 **finally**。本書直到第 10 章才會正式介紹它們。關鍵字 **try** 表示，接下來的區段（以一組大括號括起的範圍）即所謂守護區（*guarded region*），這個區段必須得到特別對待，其中之一便是，不論存在多少個 **try** 區段，守護區之後的 **finally** 子句「絕對」會被執行。（注意，在異常處理機制下，可能有許多不正常離開 **try** 區段的方式。）此處 **finally** 子句表示的是「不論發生什麼事，絕對會呼叫 **x.cleanup()**」。第 10 章會對這兩個關鍵字做更透徹的解說。

請注意，在你的 cleanup 函式中，你得留意「base class 及成員物件中的 cleanup 函式」的呼叫次序，以防範「某個子物件（subobject）與另一個子物件相依」的情形。一般而言，你應該依循 C++編譯器施加於其解構式身上的形式：首先執行你的 class 的所有特定清理動作（其次序和生成次序相反），這必須「base class 元素」尚且存活才行。然後，就像此處所示範的，呼叫 base class 的 cleanup 函式。

許多時候，「清理（cleanup）」不是問題；你只要讓垃圾回收器做事就好了。但是當你必須自行處理時，你得更加努力並且小心。

垃圾回收順序

一旦事情和「垃圾回收」有關，就不再有太多你可以信賴的事。垃圾回收器可能永遠不會被喚起；即使它被喚起，也有可能以任何它想要的次序來回收物件。因此，除了記憶體，最好不要倚賴垃圾回收機制。如果你希望發生清理動作，請自行撰寫清理用的函式，不要倚賴 **finalize()**（正如第 4 章所說，你可以強迫 Java 呼叫所有的 *finalizer*s）。

名稱遮蔽（Name hiding）

只有 C++程式員可能會對名稱遮蔽（name hiding）感到訝異，因爲在那個語言中，名稱遮蔽行爲大不相同。如果 Java 的 base class 擁有某個被重載多次的函式名稱，那麼在 derived class 中重新定義此一函式，並不會遮蔽它在 base class 中的任何版本。因此，不論該層 class 或 base class 是否定義了這個函式，都會發揮重載作用：

```
//: c06:Hide.java
// Overloading a base-class method name
// in a derived class does not hide the
// base-class versions.

class Homer {
  char doh(char c) {
    System.out.println("doh(char)");
    return 'd';
  }
  float doh(float f) {
    System.out.println("doh(float)");
    return 1.0f;
  }
}

class Milhouse {}

class Bart extends Homer {
  void doh(Milhouse m) {}
}

class Hide {
  public static void main(String[] args) {
    Bart b = new Bart();
    b.doh(1); // doh(float) used
    b.doh('x');
    b.doh(1.0f);
    b.doh(new Milhouse());
  }
} ///:~
```

你在下一章中即將看到，使用「和 base class 一模一樣的標記式（signature）及回傳型別」來覆寫（*override*）同名的函式，是再尋常不過的事了。但這種方式也很容易造成混淆（這也是 C++不允許你這麼做的原因，以杜絕你可能犯下的錯誤）。

複合與繼承之間的抉擇

Choosing composition vs. inheritance

複合與繼承，都讓你可以將子物件（subobjects）置於新的 class 中。你可能會納悶，二者之間究竟有何差異，而且何時該選用哪一種技術呢？

當你想要在新 class 中使用既有 class 的功能，而非其介面，通常可以採用複合技術。也就是說，嵌入某個物件，使你得以在新 class 中以它實現你想要的功能。但是新 class 的使用者只能看到你爲新 class 所定義的介面，不會看到內嵌的那個物件的介面。如果這正是你所希望的，你應該在新的 class 中以 **private** 形式嵌入既有的 classes 的物件。

但有時候，讓 class 使用者直接取用新 class 內的複合成份（也就是將成員物件宣告爲 **public**）是有意義的。如果各個成員物件分別完成了實作細目的隱藏，這種作法很安全。當使用者知道你組合了一堆組件，他們便更能夠輕易了解其介面。**car** 物件便是個好例子：

```
//: c06:Car.java
// Composition with public objects.

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(); // 2-door
  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);
  }
} ///:~
```

由於 car 本身的組成也是問題分析的一部份（而不僅是底層設計的一部份），所以將成員宣告爲 **public**，能夠幫助客端程式員了解此一 class 的使用方式，也降低 class 開發者所須面對的程式碼複雜度。不過你得記住，這是個特例，一般情況下，你應該將 fields（資料欄）宣告爲 **private**。

實施繼承技術時，你會使用某個既有的 class，然後開發它的一個特化版本（special version）。一般來說這代表你使用某個通用性的 class，並基於特定目的對它進行特殊化（specializing）工程。稍加思索我們就知道，以「交通工具」來合成一部車子是沒有意義的，因爲車子並非包含交通工具，車子「是一種」交通工具。這種「*is-a*（是一個）」的關係便以繼承來表達。「*has-a*（有一個）」的關係則以複合來表達。

protected（受保護的）

現在，我們已經完成了繼承的介紹。關鍵字 **protected** 終於有了意義。在理想世界中，**private** 成員應該是完全不能變通的，但實際專案中，有些時候你會希望某些東西對整個世界隱藏，而 derived classes 卻可加以存取。關鍵字 **protected** 便是這種實用主義的展現。它代表著「就此 class 的使用者來說，這是 **private**。但任何繼承自此一 class 的 derived classes，或位於同一個 package 內的其他 classes，卻可加以存取」。也就是說，Java 的 **protected** 天生具有 "friendly" 權限。

最好的準則就是，將資料成員宣告爲 **private** — 你應該絕對保留「更動底層實作」的權限。然後便可透過 **protected** 函式來控制繼承者對你所撰寫的 class 的存取權限：

```
//: c06:Orc.java
// The protected keyword.
import java.util.*;

class Villain {
  private int i;
  protected int read() { return i; }
  protected void set(int ii) { i = ii; }
  public Villain(int ii) { i = ii; }
  public int value(int m) { return m*i; }
}

public class Orc extends Villain {
  private int j;
  public Orc(int jj) { super(jj); j = jj; }
  public void change(int x) { set(x); }
} ///:~
```

你可以看到，**change()** 具有取用 **set()** 的權限，因爲它是 **protected**。

漸進式開發（Incremental development）

繼承的優點之一，便是支援漸進式開發模式。在此開發模式下，你可以加入新的程式碼，而且絕不會在既有程式碼身上衍生臭蟲。此種開發模式能夠將新的臭蟲侷限於新的程式碼中。藉由繼承既有的、基礎的 classes，並且加上新的資料成員和成員函式（並重新定義既有的函式），便可讓既有的程式碼 – 也許某人正在使用 – 不被碰觸也不含錯誤。

classes 被隔離的乾淨程度令人感到十分訝異。如果想重複運用程式碼，你甚至不需要其函式的原始碼，頂多只要匯入（import）package 就好了。這句話對繼承和複合同時成立。

你得明白，程式的開發是一個漸進過程，就像人類的學習一樣。你可以竭盡所能地分析，但是當你開始執行專案，你仍然無法知道所有解答。如果你將專案視爲一種有機、具演化能力的生物，而不是用蓋摩天大樓的方式企圖一舉完成，你便會獲得更多的成功，以及更立即的回饋。

雖然就經驗而言，繼承是個有用的技術，但是在事情進入穩定狀態之後，你得重新檢視你的 classes 階層體系，思考如何將它縮減爲更實用的結構。記住，繼承代表著一種關係的展現，它代表「這個新的 class 是一種舊的 class。」你的程式不應該只是圍繞在 bits（位元）的處理，應該透過許多不同類型的物件的生成和操作，以來自問題空間（problem space）中的術語來表現一個模型（model）。

向上轉型（Upcasting）

「繼承」技術中最重要的一個面向並非是「爲新的 class 提供函式」。繼承是介於新 class 和 base class 之間的一種關係。這種關係可以扼要地這麼說：「新 class 是既有 class 的一種形式。」

這個描述並非只是解釋「繼承」的一堆華麗辭藻，而是直接由程式語言支援的性質。假設有個名爲 **Instrument**（樂器）的 base class，其 derived class 名爲 **Wind**（管樂器）。由於繼承意謂「在 derived class 中可以使用 base class 的所有函式」，所以任何可發送給 base class 的訊息，也都可以發送給 derived class。如果 **Instrument** class 擁有 **play()**，那麼 **Wind** 也會有。這表示我們可以很精確地說，**Wind** 物件也是一種 **Instrument**。以下例子說明編譯器如何支援此種表示式：

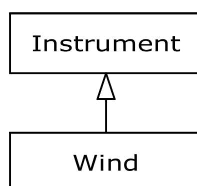
```
//: c06:Wind.java
// Inheritance & upcasting.
import java.util.*;

class Instrument {
  public void play() {}
  static void tune(Instrument i) {
    // ...
    i.play();
  }
}

// Wind objects are instruments
// because they have the same interface:
class Wind extends Instrument {
  public static void main(String[] args) {
    Wind flute = new Wind();
    Instrument.tune(flute); // Upcasting
  }
} ///:~
```

這個例子中有趣的是 **tune()**，它接受一個 **Instrument** reference。然而當 **Wind.main()** 呼叫 **tune()** 時，傳給它的卻是一個 **Wind** reference。我們知道 Java 對型別的檢查十分嚴格，而這個接受某種型別的函式，竟然可以接受另一種型別。在你明白「**Wind** 物件其實也是 **Instrument** 物件」之前，你可能會覺得這真是件怪事。此例之中沒有什麼函式是「**tune()** 可透過 **Instrument** 呼叫起來」而卻不在 **Wind** 之內的。**tune()** 函式碼可作用於 **Instrument** 以及任何衍生自 **Instrument** 的

classes 身上。這種「將 **Wind** reference 轉為 **Instrument** reference」的動作稱為「向上轉型（*upcasting*）」。

為什麼需要向上轉型（Why "upcasting"）？

這個詞彙有其歷史背景，而且根據繪製繼承階層圖時的傳統方式：將根類別（root）置於紙面上端，從上往下繪製（當然你也可以用任何你覺得有用的方式來繪製你的圖形）。因此，**Wind.java** 的繼承圖為：



從 derived class 移至 base class，在繼承圖中是向上移動，所以通常稱為「向上轉型」。向上轉型一定安全，因為這是從專用型別移至通用型別。也就是說 derived class 是 base class 的一個超集合。derived class 至少包含 base class 函式，而且可能更多。向上轉型過程中，對 class 介面造成的唯一效應是函式的「遺失」而非「獲得」。這也就是為什麼在「未曾明確表示轉型」或「未曾指定特殊標記」的情況下，編譯器仍然允許向上轉型的原因。

你也可以執行和向上轉型方向相反的「向下轉型（*downcasting*）」，但這會牽扯到一個與第 12 章討論課題有關的難題。

複合（Composition）vs. 繼承（inheritance），再探

物件導向編程中，產生及運用程式碼的最可能形式，便是將資料和函式包裝起來成為 class，然後運用其物件。你可能透過「複合」，以既有的 class 開發出新的 classes。動用「繼承」的頻率比較少。所以雖然 OOP 的學習過程中常常強調繼承，但並不代表你應該處處使用它。

相反地，你應該謹慎使用它：只有在明顯能夠產生實用價值時，才使用繼承。究竟應該使用複合或繼承，最清楚的判斷方式就是問你自己，是否需要將新的 class 向上轉型爲 base class。如果你必須向上轉型，你就應該使用繼承。如果不需要向上轉型，那麼你應該仔細考慮是否需要動用繼承。下一章（polymorphism，多型）將提出向上轉型的一個最具競爭力的理由。如果你時時詢問自己：「我需要向上轉型嗎？」，那麼，當你面對複合或繼承時，你便擁有一個極佳的判斷工具。

關鍵字 final

如果望文生義，可能會對 Java 的關鍵字 **final** 有所誤解。一般來說其意思是：「這是不能被改變的」。是的，基於設計和效率兩大理由你可能希望阻止改變。這兩個理由十分不同，因此可能造成對關鍵字 **final** 的誤用。

以下各節探討可以使用 **final** 的三個地方：data、methods、classes。

Final data

許多程式語言都提供某種機制，用來告訴編譯器某塊資料是「固定不變的」。不變的資料可能很有用，因爲它：

1. 可以是永不改變的「編譯期常數（*compile-time constant*）」。
2. 可以在執行期（*run-time*）被初始化，而你卻不想再改變它。

以編譯期常數而言，編譯器可以將這個常數包進任何一個用到它的計算式中；也就是說，編譯期間就能夠執行某些計算，減少執行期的負擔。在 Java 中，此類常數必須屬於基本型別，而且必須以關鍵字 **final** 修飾之。定義此類常數的同時，必須給定其值。

如果某一筆資料既是 **static** 也是 **final**，那麼它會擁有一塊無法改變的儲存空間。

將 **final** 用於 object references 而不用於基本型別時，其實際意義可能有點令人困惑。用於基本型別時，**final** 讓 value（數值）保持不變，但是用

於 object reference 時，**final** 讓 reference 保持不變。某個 reference 一旦被初始化用以代表某個物件之後，便再也不能改而指向另一個物件。但此時物件本身的內容卻是可以更動的；Java 並未提供「讓任何物件保持不變」的機制（不過你還是可以撰寫你自己的 class，產生令物件保持不變的效果）。此一限制對同樣身為物件的 array 而言，也是成立的。

下面是個例子，用來示範 **final** 資料成員：

```
//: c06:FinalData.java
// The effect of final on fields.

class Value {
  int i = 1;
}

public class FinalData {
  // Can be compile-time constants
  final int i1 = 9;
  static final int VAL_TWO = 99;
  // Typical public constant:
  public static final int VAL_THREE = 39;
  // Cannot be compile-time constants:
  final int i4 = (int)(Math.random()*20);
  static final int i5 = (int)(Math.random()*20);

  Value v1 = new Value();
  final Value v2 = new Value();
  static final Value v3 = new Value();
  // Arrays:
  final int[] a = { 1, 2, 3, 4, 5, 6 };

  public void print(String id) {
    System.out.println(
      id + ": " + "i4 = " + i4 +
      ", i5 = " + i5);
  }
  public static void main(String[] args) {
    FinalData fd1 = new FinalData();
    //! fd1.i1++; // Error: can't change value
    fd1.v2.i++; // Object isn't constant!
    fd1.v1 = new Value(); // OK -- not final
```

```
    for(int i = 0; i < fd1.a.length; i++)
      fd1.a[i]++; // Object isn't constant!
    //! fd1.v2 = new Value(); // Error: Can't
    //! fd1.v3 = new Value(); // change reference
    //! fd1.a = new int[3];

    fd1.print("fd1");
    System.out.println("Creating new FinalData");
    FinalData fd2 = new FinalData();
    fd1.print("fd1");
    fd2.print("fd2");
  }
} ///:~
```

由於 **i1** 和 **VAL_TWO** 都是帶有編譯期數值的 **final** 基本型別，所以它們都可以被當做編譯期常數來使用，沒有什麼重大區別。**VAL_THREE** 的定義方式是更爲典型的常數定義方式：定義爲 **public**，所以可被用於 package 之外；定義爲 **static**，強調它只有一份；定義爲 **final**，宣告它是個常數。請注意，帶有常數初值的 **final static** 基本型別，習慣上以底線來隔開字與字（這個習慣濫觴於 C 語言）。同時也請注意，**i5** 的值在編譯期無法得知，所以並未以大寫字母表示。

我們不能只因爲某筆資料被宣告爲 **final**，就認爲在編譯期便知其值。上述程式爲了示範這一點，在執行期隨機產生的數字做爲 **i4** 和 **i5** 的初值。這也說明了將 **final** 數值宣告爲 **static** 和宣告爲 non-**static** 的差別。這個差別只有「當其數值係在執行期被初始化」時才會顯現，因爲編譯器對待任何編譯期數值（compile-time values）的態度都是一樣的（而且它們可能因爲最佳化而消失掉）。這個差別可以從某些執行結果觀察出來：

```
fd1: i4 = 15, i5 = 9
Creating new FinalData
fd1: i4 = 15, i5 = 9
```

```
fd2: i4 = 10, i5 = 9
```

請注意，**fd1** 和 **fd2** 中的 **i4** 值皆不相同，但 **i5** 值不會因爲產生了第二個 **FinalData** 物件而改變。這是因爲它是 **static**，而且只有在載入 class 時才進行初始化，不會在每次產生新物件時都再被初始化一次。

從 **v1** 到 **v4** 的這幾個變數，說明了 **final** reference 的意義。正如你在 **main()** 中觀察到的，不能因爲 **v2** 是 **final**，就認爲你無法改變其值。不過，你無法將 **v2** 重新指向另一個物件，因爲 **v2** 是 **final**。這也正是 **final** 用於 reference 時的意義。你會發現，同樣的意義對 array 來說依然成立，因爲它不過也只是另一種 reference 罷了。不過就我所知，沒有任何方法可以令 array references 本身成爲 **final**。將 references 宣告爲 **final** 似乎比將基本型別宣告爲 **final** 不實用多了。

Blank finals

Java 允許產生所謂「留白的 finals」（*blank final*），也就是允許我們將資料成員宣告爲 **final**，卻不給予初值。任何情況下，blank finals 必須在使用之前進行初始化，而且編譯器會保證此事。不過 blank finals 對於 **final** 關鍵字的使用提供了更多彈性，因爲這麼一來，class 內的 **final** 資料成員便可以在每個物件中有所不同，但依舊保持其「恆長不變」的特性。以下便是一例：

```
//: c06:BlankFinal.java
// "Blank" final data members.

class Poppet { }

class BlankFinal {
  final int i = 0; // Initialized final
  final int j; // Blank final
  final Poppet p; // Blank final reference
  // Blank finals MUST be initialized
  // in the constructor:
  BlankFinal() {
    j = 1; // Initialize blank final
    p = new Poppet();
  }
```

```
BlankFinal(int x) {
    j = x; // Initialize blank final
    p = new Poppet();
  }
  public static void main(String[] args) {
    BlankFinal bf = new BlankFinal();
  }
} ///:~
```

編譯器強迫你一定得對所有 **final**s 執行賦值動作 — 如果不是發生在其定義處，就得在每個建構式中以運算式設定其值。這也就是爲什麼能夠保證「**final** 資料成員在被使用前絕對會被初始化」的原因。

Final arguments

Java 允許你將引數（arguments）宣告爲 **final**，只要在引數列中宣示即可，意謂你無法在此函式中令該引數（一個 reference）改指它處：

```
//: c06:FinalArguments.java
// Using "final" with method arguments.

class Gizmo {
  public void spin() {}
}

public class FinalArguments {
  void with(final Gizmo g) {
    //! g = new Gizmo(); // Illegal -- g is final
  }
  void without(Gizmo g) {
    g = new Gizmo(); // OK -- g not final
    g.spin();
  }
  // void f(final int i) { i++; } // Can't change
  // You can only read from a final primitive:
  int g(final int i) { return i + 1; }
  public static void main(String[] args) {
    FinalArguments bf = new FinalArguments();
    bf.without(null);
    bf.with(null);
  }
```

```
} ///:~
```

請注意，你仍然可以將 **null** reference 當做 **final** 引數，就像面對 non-**final** 引數一樣。是的，這麼做並不會帶來編譯上的困擾。

f() 和 **g()** 這兩個函式說明了「當基本型別引數為 **final**」時會發生什麼事：你可以讀取引數所代表的值，但無法改變該值。

Final methods

使用 **final** methods（函式）的原因有二。第一，鎖住這個函式，使 derived class 無法改變其意義。這是基於設計的一種考量：也許你希望確保某個函式的行為在繼承過程中保持不變，而且無法被覆寫（overridden）。

第二個原因是效率。如果你將某個函式宣告為 **final**，等於允許編譯器將所有對此函式的呼叫動作轉化為 *inline*（行內）呼叫。當編譯器看到一個 **final** 函式呼叫動作，它可以（根據它自己的謹慎判斷）不採用正常對待方式（亦即插入某段程式碼以執行「函式呼叫機制」：將引數送入 stack、跳至函式程式碼所在位置並執行、跳回並清除 stack 內的引數、處理回傳值），而以函式本體取代那個呼叫動作。這麼做能夠降低函式呼叫動作所引發的額外負擔。當然，如果函式體積龐大，inlining 會迫使整個程式碼大小隨之膨脹。而且你可能無法藉此獲得效能的改善，是的，你獲得的效能利益將因為「花費在函式中的時間」而顯得不成比例。這意味 Java 編譯器可以偵測這些情況，並聰明地決定是否對 **final** 函式執行 inline 動作。不過，最好不要相信你的編譯器具備此種才華，最好是在某個函式真的體積很小或是你真的不希望它被覆寫時，才將它宣告為 **final**。

final 和 private

class 中的所有 **private** 函式自然而然會是 **final**。因為你無法取用 **private** 函式，當然也就無從覆寫（即使當你嘗試覆寫它而編譯器沒有給出任何錯誤訊息，你仍然沒有覆寫成功。實際上你是產生了一個新的

函式）。你可以將關鍵字 **final** 加於 **private** 函式身上，但這麼做不會帶來任何額外意義。

這個問題可能引發人們的困惑，因爲如果你試著覆寫某個 **private** 函式（隱隱有 **final** 味道），似乎也可以：

```
//: c06:FinalOverridingIllusion.java
// It only looks like you can override
// a private or private final method.

class WithFinals {
  // Identical to "private" alone:
  private final void f() {
    System.out.println("WithFinals.f()");
  }
  // Also automatically "final":
  private void g() {
    System.out.println("WithFinals.g()");
  }
}

class OverridingPrivate extends WithFinals {
  private final void f() {
    System.out.println("OverridingPrivate.f()");
  }
  private void g() {
    System.out.println("OverridingPrivate.g()");
  }
}

class OverridingPrivate2
  extends OverridingPrivate {
  public final void f() {
    System.out.println("OverridingPrivate2.f()");
  }
  public void g() {
    System.out.println("OverridingPrivate2.g()");
  }
}

public class FinalOverridingIllusion {
```

```
  public static void main(String[] args) {
    OverridingPrivate2 op2 =
      new OverridingPrivate2();
    op2.f();
    op2.g();
    // You can upcast:
    OverridingPrivate op = op2;
    // But you can't call the methods:
    //! op.f();
    //! op.g();
    // Same here:
    WithFinals wf = op2;
    //! wf.f();
    //! wf.g();
  }
} ///:~
```

「覆寫」（overriding）只能夠發生在「函式屬於 base class 介面」時。也就是說，你必須能夠將某個物件向上轉型至其基本型別，並呼叫同一個（同名）函式（下一章會對此點做更進一步的釐清）。如果某個函式是 **private**，它便不涵括於 base class 的介面中。它只不過是隱藏在 class 內的某段程式碼，而恰好具有那個名字罷了。即使你在 derived class 中撰寫了某個 **public** 或 **protected** 或 "friendly" 函式，對它而言也並不就因此具備「剛好擁有 base class 中的某個名稱」的關聯性。由於 **private** 函式無法接觸，有效隱藏，所以它不需要被任何事物納入考量 – 它只是在它所棲身的 class 中因程式碼的組織而存在。

Final classes

當你將整個 class 宣告為 **final**（將關鍵字 **final** 置於 class 定義式前），等於宣告你並不想繼承此一 class，也不允許別人這麼做。換句話說，或許基於 class 設計上的考量，你不需要再有任何更動；或許基於安全保密的考量，你不希望繼承動作發生。你也許會想到效率議題，你應該確保與此 class objects 有關的任何活動，都儘可能地有效率。

```
//: c06:Jurassic.java
// Making an entire class final.
```

```
class SmallBrain {}

final class Dinosaur {
  int i = 7;
  int j = 1;
  SmallBrain x = new SmallBrain();
  void f() {}
}

//! class Further extends Dinosaur {}
// error: Cannot extend final class 'Dinosaur'

public class Jurassic {
  public static void main(String[] args) {
    Dinosaur n = new Dinosaur();
    n.f();
    n.i = 40;
    n.j++;
  }
} ///:~
```

請注意，不論 class 是否被定義爲 **final**，資料成員既可以是 **final**，也可以不是，視你的意志而定。**final** data 的原始規則仍然適用。將 class 定義爲 **final**，只不過是要杜絕繼承。不過，由於這麼做會阻止繼承動作，所以 **final** class 中的所有函式也都自然是 **final**，因爲沒有人能夠加以覆寫。如果你明確地將某個函式宣告爲 **final**，編譯器一樣具有效率上的選擇權。

你可以將關鍵字 **final** 加於 **final** class 內的函式之前，但這麼做並不會增加任何意義。

最後的告誡

當你設計 class 時，將函式宣告爲 **final** 可能是合理的。你可能認爲使用 class 時效率很重要，而且不應該有任何人覆寫你的函式。某些時候這是對的。

但請格外小心你所做的假設。一般來說我們很難預先考慮 class 可能被重複運用的方式，對通用性的 class 來說尤其如此。如果你將某個函式定義為 **final**，你可能會阻礙「其他程式員在另一個專案中，透過繼承，重複運用此一 class」的可能性。因為你沒有想到它會被這麼運用。

Java 標準程式庫便是一個極佳的例子。特別是在 Java 1.0/1.1 中被廣泛使用的 **Vector** class，從它的名字看來，便知道其設計目的在於效率。如果它的所有函式都未被宣告為 **final**，它可能會更為有用。你可能想要繼承如此有用的 base class 並加以覆寫（override）。這種想法很容易理解。但是不知怎的，其設計者卻認為這麼做不妥當。這裡有兩個出人意表的原因。第一，**Stack** 繼承自 **Vector**，意味 **Stack** 是個 **Vector**，事實上這不是正確的思考邏輯。第二，**Vector** 的許多重要函式，例如 **addElement()** 和 **elementAt()**，都是 **synchronized**。第 14 章會告訴你，這造成極大的效能負擔，或許會因而抵消因 **final** 而得到的所有效能改善。這種情況使人們更加確信這一句話：程式員很難正確揣測最佳化動作應該發生於何處。這麼拙劣的設計，竟然被放在我們大家都必須使用的標準程式庫中，真是夠慘的了。幸好 Java 2 的容器相關程式庫以 **ArrayList** 汰換了 **Vector**。**ArrayList** 的行為合理多了，不幸的是許多新開發的程式仍然使用那些老舊的容器程式庫。

Hashtable 亦頗值得一書。它是標準程式庫中的另一個 class，但未具備任何 **final** 函式。一如本書它處所言，有些 classes 很明顯是由一群完全不相干的人設計出來（你會發現 **Hashtable** 的函式名稱相對於 **Vector** 而言簡短多了，這是另一個證據）。對 class library 使用者來說，這又是另一種不該如此輕率的東西。規則的不一致，只會讓使用者付出更多額外工夫。這是對草率設計和草率撰碼的另一個驚嘆。請注意，Java 2 的容器程式庫已經以 **HashMap** 汰換了 **Hashtable**。

初始化與 class 的載入

有許多傳統語言，其程式在起動（startup）過程中便會全部被載入，緊接著初始化動作，然後便開始執行。在這些語言中，初始化過程必須被小心翼翼地控制，才能確保 **statics** 的初始化順序不會引發問題。以 C++為例，如果某個 **static** 預期另一個 **static**「在初始化前能夠被正常使用」，便會發生問題。

Java 沒有這種問題，因爲它採用另一種載入方式。由於 Java 中的每樣事物都是物件，許多動作就變得更爲簡單，載入動作亦如是。下一章會讓你學得更完整。每個 class 經過編譯之後，存於專屬的個別檔案中。這個檔案只在必要時才被載入。一般而言你可以這麼說：「class 程式碼在初次被使用時才被載入」。所謂初次被使用，不僅是其第一個物件被建構之時，也可能是在某個 **static** 資料成員或 **static** 函式被取用時。

「首次使用 class」的時間點，也正是靜態初始化（**static** initialization）的進行時機。任何 **static** 物件和 **static** 程式區段被載入時，都會依據它們在程式碼中的次序（也就是它們在 class 定義式中的出現次序）加以初始化。當然，**statics** 只會被初始化一次。

繼承與初始化

仔細檢視包括繼承在內的一整個初始化過程，對於其中因果關係的了解極有幫助。請看以下程式碼：

```
//: c06:Beetle.java
// The full process of initialization.

class Insect {
  int i = 9;
  int j;
  Insect() {
    prt("i = " + i + ", j = " + j);
```

```
    j = 39;
  }
  static int x1 =
    prt("static Insect.x1 initialized");
  static int prt(String s) {
    System.out.println(s);
    return 47;
  }
}

public class Beetle extends Insect {
  int k = prt("Beetle.k initialized");
  Beetle() {
    prt("k = " + k);
    prt("j = " + j);
  }
  static int x2 =
    prt("static Beetle.x2 initialized");
  public static void main(String[] args) {
    prt("Beetle constructor");
    Beetle b = new Beetle();
  }
} ///:~
```

這個程式的輸出結果是：

```
static Insect.x1 initialized
static Beetle.x2 initialized
Beetle constructor
i = 9, j = 0
Beetle.k initialized
k = 47
j = 39
```

當你執行 **Beetle** 時，首先發生的動作便是企圖取用 **Beetle.main()**（這是一個 **static** 函式），於是載入器被啓動，找出 **Beetle** class 編譯過的程式碼（應該被置於名爲 **Beetle.class** 的檔案中）。載入過程中由於關鍵字 **extends** 的存在，載入器得知這個 class 擁有 base class，於是接續載入。無論你是否產生 base class 的物件，這個動作都會發生。請試著將其「物件生成空間」註解掉，藉此證明這一點。

如果 base class 還有 base class，那麼便會接續載入第二個 base class，依此類推。接下來，root base class（本例爲 **Insect**）中的靜態初始化動作會被執行，然後是其 derived class，依此類推。上述方式很重要，因爲 derived class 的靜態初始化動作是否正確，可能和 base class 的成員是否被正確初始化有關。

至止，所有必要的 classes 都已被載入，可以開始產生物件了。首先，物件內的所有基本型別都會被設予預設值，object reference 則被設爲 **null** — 也就是說將物件記憶體都設爲二進位零值。然後，base class 的建構式會被喚起。本例之中它會被自動喚起，但你也可以使用 **super**（做爲 **Beetle()** 建構式的第一個動作），自行呼叫 base class 的某個建構式。base class 的建構過程和其 derived class 建構式中的次序相同。base class 建構式完成之後，其「instance 變數」（譯註：亦即資料成員；相對於「class 變數」）會以其出現次序被初始化。最後才執行建構式本體的剩餘部份。

摘要

繼承和複合都允許你根據既有的型別產生新型別。一般而言你會使用「複合」來重複運用既有型別，使其成爲新型別底層實作的一部份；至於「繼承」則用於介面的重複運用。由於 derived class 具備了 base class 的介面，所以它可以「向上轉型（*upcast*）」至 base class，這對多型（polymorphism）來說極爲重要，詳見下一章。

儘管物件導向編程（OOP）極爲強調「繼承」，但你一開始進行設計時，應該先考慮使用「複合」，只有在明確必要時才使用「繼承」。「複合」較具彈性，而經由「繼承」帶來的靈巧性，你可以在執行期改變成員物件的實際型別和行爲，如此一來便可以在執行期改變複合而成的物件的行爲。

雖然，透過複合和繼承達到程式碼的重複運用，對於快速專案開發來說極具助益，但是在其他程式員使用之前，通常還是得再一次設計你的 class 階層架構。我們的努力目標是：讓階層架構中的每個 class 各自具備特定用途，而且既不太大（包含太多功能，難以重複運用）也不太小（無法在不加入其他功能的情況下單獨使用之）。

練習

某些經過挑選的題目，其解答置於《*The Thinking in Java Annotated Solution Guide*》電子文件中。僅需小額費用便可自 *www.BruceEckel.com*.網站取得。

1. 請撰寫兩個帶有 *default* 建構式（引數列空白）的 classes **A** 和 **B**。請繼承自 **A**，產生一個 class **C**，並在 **C** 中以 **B** 物件做爲成員。請不要爲 **C** 提供任何建構式。現在，產生一個 **C** 物件，並觀察結果。

2. 修改上題程式，使 **A** 和 **B** 都具有帶引數的建構式，而非 *default* 建構式。爲 **C** 撰寫一個建構式，並在其中執行所有初始化動作。

3. 先撰寫一個簡單的 class，並在撰寫第二個 class 時，將某個資料成員定義爲第一個 class 的物件。請使用緩式初始化（lazy initialization）來產生這個物件實體。

4. 繼承 class **Detergent**，產生一個新的 class。覆寫其 **scrub()** 並加入一個名爲 **sterilize()** 的新的函式。

5. 使用 **Cartoon.java**，並將 **Cartoon** class 的建構式註解掉。請解釋所發生的現象。

6. 使用 **Chess.java**，並將 **Chess** class 的建構式註解掉。請解釋所發生的現象。

7. 請證明爲你而產生的 *default* 建構式是由編譯器合成的。

8. 請證明 base class 的建構式 (a) 絕對會被喚起、(b) 在 derived class 建構式之前被喚起。

9. 撰寫一個 base class，令它僅具有一個 non-*default* 建構式。再撰寫一個 derived class，令它同時具備 *default* 建構式和 non-

default 建構式。請在 derived class 建構式中呼叫 base class 的建構式。

10. 撰寫一個 **Root** class，並令它分別含有以下 class 的實體（亦由你來撰寫）： **Componenet1**、**Component2**、**Component3**。現在，從 **Root** 衍生出 class **Stem**，並令它含有上述各個 "component"。所有 classes 都應該具備一個能夠印出 class 相關訊息的 *default* 建構式。

11. 修改上題的程式，令每個 class 僅有 non-*default* 建構式。

12. 將 **cleanup()** 適當加至練習 11 中的所有 classes。

13. 撰寫某個 class，令它擁有一個重載三次的函式。爲它衍生一個新的 class，並爲上述函式 加入一份新的重載定義。證明這四個函式 在 derived class 中都可用。

14. 在 **Car.java** 中，請將 **service()** 加至 class **Engine**，並在 **main()** 中呼叫它。

15. 撰寫位於 package 內的某個 class，並令它含有一個 **protected** 函式。在此 package 之外，試著呼叫該 **protected** 函式，並解釋其結果。接著，繼承剛才那個 class，並在 derived class 的某個函式中呼叫上述的 **protected** 函式。

16. 撰寫一個 **Amphibian** class，並令 **Frog** 繼承之。請將適當的函式置於 base class 中。並在 **main()** 中產生 **Frog** 物件，且向上轉型至 **Amphibian**。示範所有的函式都能夠有效運作。

17. 修改練習 16 的程式，令 **Frog** 覆寫其 base class 中的函式的定義（以同樣的函式標記式來撰寫新的定義）。請留意 **main()** 所發生的事。

18. 請撰寫一個 class，令它擁有 **static final** 資料成員和 **final** 資料成員，並說明二者的差異。

19. 請撰寫一個 class，令它擁有 blank **final** reference。請在你使用這個 reference 之前，於某個函式（但非建構式）中執行該 blank **final** 的初始化動作。請說明以下保證：**final** 被使用之前一定會被初始化，而且一經初始化便無法改變。

20. 請撰寫一個 class，令它擁有 **final** 函式。衍生一個新的 class 並覆寫該函式。

21. 請撰寫一個 **final** class，並試著加以繼承。

22. 請證明，class 載入動作只會發生一次。也請證明，class 載入動作可能發生在 class 的第一個實體誕生時，或其 **static** 成員第一次被取用時。

23. 在 **Beetle.java** 中，試著從 class **Beetle** 繼承出一種特殊的甲蟲，並令它依循既有 classes 的相同形式。請追蹤輸出結果，並試著加以解釋。

7: 多型 Polymorphism

除了資料抽象化（data abstraction）與繼承（Inheritance）以外，物件導向編程語言的第三個核心本質便是多型（polymorphism）。

多型提供了「介面與實作分離」的另一個重要性，能將 *what*（是什麼）自 *how*（怎麼做）之中抽離。多型不但能改善程式的組織架構及可讀性，更能開發出「可擴充」的程式。具備此種特質的程式，不僅能夠在原本的專案開發過程中逐漸成長，也能藉由增加新功能來擴充規模。

封裝（*encapsulation*）藉著「將特徵（*characteristics*）與行爲（*behaviors*）結合在一起」而產生新的資料型別。實作隱藏（*implementation hidden*）則藉由「將細目（details）宣告爲 **private**」而分離出介面（*interface*）與實作（*implementation*）。此類機制對於擁有程序式（procedural）設計背景的人們來說，是有意義的。但多型（*polymorphism*）卻打算除去型別之間的耦合關係。前一章你看到了，繼承機制允許你不但能將某個物件以其本身型別視之，亦能以其基礎型別（base type）視之。此一能力極爲重要，有了這種能力，我們便能夠將多個型別視爲同一個型別，而一份程式碼也因此得以同時作用於這些（其實並不相同的）型別之上。*polymorphical method call*（多型函式呼叫）使某個型別有能力表現它與另一個相似型別的區別 – 只要這兩個型別皆衍生自相同的基礎型別（base type）。相似型別之間的區別可藉由「函式的行爲差異」來展現，而這些函式都可透過 base class 喚起。

本章中，你會透過一些簡單範例（焦點全放在多型行爲身上）學習有關多型（也稱爲動態繫結 *dynamic binding*、後期繫結 *late binding*、或執行時期繫結 *run-time binding*）的種種事物。

再探向上轉型（Upcasting）

你已經在第 6 章看到了，物件既能以其自身型別的形式被使用，又能被視爲其基礎型別而被使用。「將某個 object reference 視爲一個 reference to

base type」的動作，稱爲「向上轉型（*upcasting*）」，之所以這麼說，是因爲我們習慣在繼承樹狀圖中將 base class 置於上端。

在此同時，你也會看到伴隨而來的問題。下面是一個具體實例：

```
//: c07:music:Music.java
// Inheritance & upcasting.

class Note {
  private int value;
  private Note(int val) { value = val; }
  public static final Note
    MIDDLE_C = new Note(0),
    C_SHARP  = new Note(1),
    B_FLAT   = new Note(2);
} // Etc.

class Instrument {
  public void play(Note n) {
    System.out.println("Instrument.play()");
  }
}

// Wind objects are instruments
// because they have the same interface:
class Wind extends Instrument {
  // Redefine interface method:
  public void play(Note n) {
    System.out.println("Wind.play()");
  }
}

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
  }
```

```
} ///:~
```

Music.tune() 接受一個 **Instrument** reference，但也接受所有衍生自 **Instrument** 的 classes。你可以在 **main()** 中觀察到這個特性，當 **Wind** 被傳入 **tune()** 時，無需任何轉型動作。這麼做是可以的，因爲 **Wind** 繼承自 **Instrument**，所以 **Instrument** 的介面必定也存在於 **Wind** 中。自 **Wind** 向上轉型至 **Instrument** 可能會「窄化」其介面，但無論如何不會窄於 **Instrument** 的介面。

將物件的型別忘掉

你可能覺得這個程式頗爲奇怪。爲什麼有人要刻意把物件的型別忘掉呢？這種情況發生於「向上轉型」之時。其實「讓 **tune()** 接收 **Wind** reference」或許更爲直觀，但如果你那麼做，你就得爲系統中的每個 **Instrument** 型別重新撰寫 **tune()**。如果我們採用那種作法，並加入 **Stringed**（弦樂器）與 **Brass**（銅管樂器）：

```
//: c07:music2:Music2.java
// Overloading instead of upcasting.

class Note {
  private int value;
  private Note(int val) { value = val; }
  public static final Note
    MIDDLE_C = new Note(0),
    C_SHARP = new Note(1),
    B_FLAT = new Note(2);
} // Etc.

class Instrument {
  public void play(Note n) {
    System.out.println("Instrument.play()");
  }
}

class Wind extends Instrument {
  public void play(Note n) {
    System.out.println("Wind.play()");
```

```
  }
}

class Stringed extends Instrument {
  public void play(Note n) {
    System.out.println("Stringed.play()");
  }
}

class Brass extends Instrument {
  public void play(Note n) {
    System.out.println("Brass.play()");
  }
}

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); // No upcasting
    tune(violin);
    tune(frenchHorn);
  }
} ///:~
```

這麼寫當然行得通，但主要的缺點是：你得爲你新加入的每個 **Instrument** class 撰寫專屬的函式。這代表一開始你得花不少編程功夫，這也代表如果你想加入和 **tune()** 類似的新的函式，或是加入衍生自 **Instrument** 的新型別，得花不少力氣。此外，如果你忘了重新定義函

式，編譯器不會給你任何錯誤訊息，因而使得型別的整個使用過程變得更難以管理。

如果我們能夠只寫一份函式，接收 base class（而非任何特定的 derived class）作爲引數，事情會變得更好嗎？也就是說，如果能夠忘掉 derived class 的存在，只撰寫與 base class 溝通的程式碼，會不會比較好？

這正是多型允許你做的事。然而，大多數擁有程序式（procedural）設計背景的程式員，對於多型的運作方式一開始都會有許多困惑。

竅門

Music.java 的困難點，在我們執行該程式之後便可發現。其輸出結果顯示 **Wind.play()** 會被執行。這無疑是我們想要的，但這個結果卻似乎不太合理。請看 **tune()**：

```
public static void tune(Instrument i) {
  // ...
  i.play(Note.MIDDLE_C);
}
```

它接收一個 **Instrument** reference。那麼，在這個例子中，編譯器如何才有可能知道此 **Instrument** reference 指向的是 **Wind** 而非 **Brass** 或 **Stringed** 呢？編譯器無能爲力。想要對此課題做更深入的了解，必須仔細檢驗繫結（*binding*）這個主題。

Method-call（函式呼叫）繫結方式

所謂「繫結（*binding*）」，就是建立 method call（函式呼叫）和 method body（函式本體）的關聯。如果繫結動作發生於程式執行前（由編譯器和連結器完成），稱爲「先期繫結（*early binding*）」。以往你可能從未聽過這個名詞，因爲程序式語言沒有其他選擇，一定是先期繫結。C 編譯器只有一種 method call，就是先期繫結。

前述範例程式之所以令人困惑，完全是因爲先期繫結。因爲，當編譯器手上只握有一個 **Instrument** reference 時，它無法得知究竟該呼叫哪一個函式。

解決方法便是透過所謂的後期繫結（*late binding*）：繫結動作將在執行期才根據物件型別而進行。後期繫結也被稱爲執行期繫結（*run-time binding*）或動態繫結（*dynamic binding*）。程式語言若欲實作出後期繫結，必須具備「得以在執行期判知物件型別」並「呼叫其相應之函式」的機制。也就是說，編譯器仍然不知道物件的型別，但「method call 機制」會找出正確的 method body 並加以呼叫。程式語言的後期繫結機制作法因人而異，但是你可以想像，必得有某種「型別資訊」被置於物件內。

Java 的所有函式，除了被宣告爲 **final** 者，皆使用後期繫結。這意味在一般情況下，你不需要判斷後期繫結動作何時發生 – 它會自動發生。

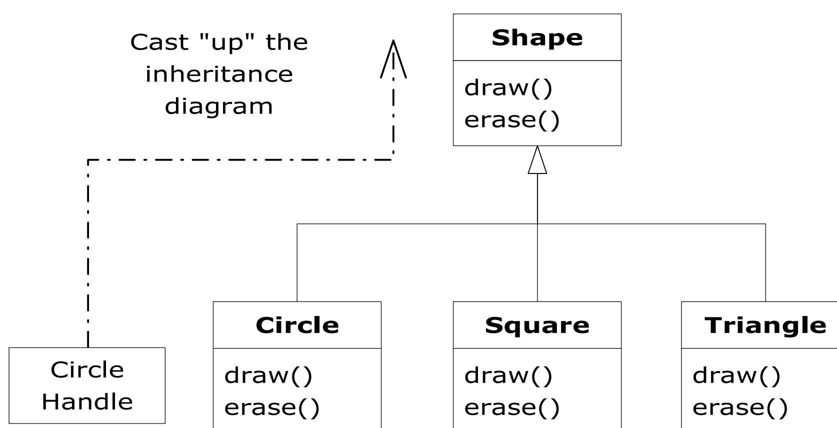
爲什麼會想將某個函式宣告爲 **final** 呢？一如前一章所述，這是爲了防止其他人覆寫該函式。但或許更重要的是，這麼做可以「關閉」動態繫結。或者說，這麼做便是告訴編譯器：動態繫結是不需要的。於是編譯器便得以爲 **final** method call 產生效率較佳的程式碼。不過大多數時候這麼做並不會爲你的程式帶來整體效能提昇。所以最好是基於設計上的考量來決定是否使用 **final**，而不要企圖藉由它來改善效能。

產生正確的行為

一旦你知道所有 Java 函式都是透過後期繫結達到多型性之後，你便可以撰寫與 base class 溝通的程式碼。而且你也明白，同一份程式碼在面對所有 derived classes 時皆能運作無誤。另一個說法是：只要「發送訊息給某個物件，讓該物件自行找到應該做的事」就好了。

OOP 中最經典的例子，莫過於「形狀（shape）」了。因爲它極易被視覺化，所以被廣泛採用。不幸的是這個例子容易誤導程式員以爲 OOP 僅能用於圖形化程式設計，那當然不是真的。

在 shape 範例程式中，有個名爲 **Shape** 的 base class，以及許多不同的 derived class：**Circle**、**Square**、**Triangle** 等等。我們可以說「圓是一種形狀」，這話極易理解。這便是這個例子之所以好的原因。classes 繼承圖顯示了它們之間的關係：



向上轉型可以發生在這麼簡單的一行述句中：

```
Shape s = new Circle();
```

這裡產生了一個 **Circle** 物件，所得的 reference 立即被指派給 **Shape**。指派（賦值）動作看似錯誤（將某個型別指派至另一型別），但因爲 **Circle** 是一個 **Shape**，這麼做絲毫沒有問題。所以，編譯器接受此行述句而不發出錯誤訊息。

假設你呼叫某個 base class 函式（它被 derived class 覆寫）：

```
s.draw();
```

你也許會以爲是 **Shape** 的 **draw()** 被喚起，因爲 **s** 終究是個 **Shape** reference。編譯器如何能夠得知其他任何事情呢！然而，由於後期繫結（多型），最終還是正確喚起了 **Circle.draw()**。

下面這個例子有一點變化：

```
//: c07:Shapes.java
```

```
// Polymorphism in Java.

class Shape
  void draw() {}
  void erase() {}
}

class Circle extends Shape {
  void draw()
    System.out.println("Circle.draw()");
  }
  void erase()
    System.out.println("Circle.erase()");
  }
}

class Square extends Shape {
  void draw()
    System.out.println("Square.draw()");
  }
  void erase()
    System.out.println("Square.erase()");
  }
}

class Triangle extends Shape {
  void draw()
    System.out.println("Triangle.draw()");
  }
  void erase()
    System.out.println("Triangle.erase()");
  }
}

public class Shapes {
  public static Shape randShape() {
    switch((int)(Math.random() * 3)) {
      default:
      case 0: return new Circle();
      case 1: return new Square();
      case 2: return new Triangle();
```

```
    }
  }
  public static void main(String[] args) {
    Shape[] s = new Shape[9];
    // Fill up the array with shapes:
    for(int i = 0; i < s.length; i++)
      s[i] = randShape();
    // Make polymorphic method calls:
    for(int i = 0; i < s.length; i++)
      s[i].draw();
  }
} ///:~
```

base class **Shape** 建立的是繼承自 **Shape** 的所有 classes 的共同介面 – 也就是說，所有形狀都能被繪製（*draw*）和擦拭（*erase*）。derived class 藉由覆寫這些定義而提供自己（特定形狀）的獨特行爲。

main class **Shapes** 含有一個 **static randShape()**。每次呼叫它，它便產生並回傳一個 reference，指向隨機挑選的一個 **Shape** 物件。請注意，其中每一個 **return** 述句，無論將「指向 **Circle** 或 **Square** 或 **Triangle**」之 reference 回傳，都會發生向上轉型，使型別轉爲 **Shape**。所以當你呼叫這個函式，你永遠無法察知你所獲得的物件的實際型別究竟爲何，因爲你永遠只會拿到一個（被一般化了的）**Shape** reference。

main() 內含一個由 **Shape** reference 組成的 array，並透過 **randShape()** 填入元素。此時，你知道你擁有一些 **Shape**s，但你不知道任何更細部的事情（編譯器也不知道）。不過當你一一走訪 array 元素，並呼叫每個 **Shape** 的 **draw()** 時，各型別的專屬行爲竟然神奇地發生了。這可以從執行結果得知：

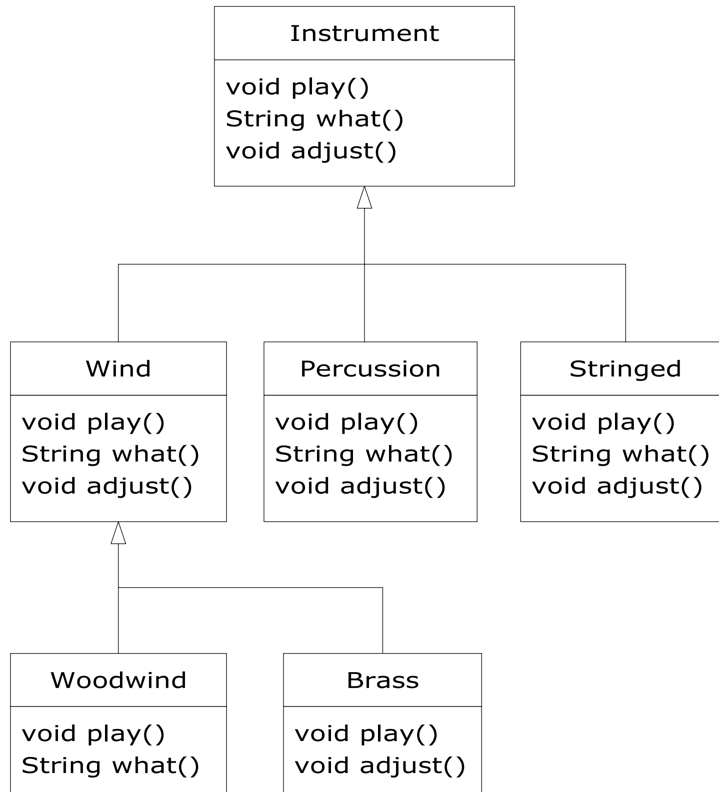
```
Circle.draw()
Triangle.draw()
Circle.draw()
Circle.draw()
Circle.draw()
Square.draw()
Triangle.draw()
Square.draw()
Square.draw()
```

當然，由於每次執行都是隨機挑選不同的形狀，所以你會得到不同的執行結果。採用隨機挑選，是爲了更透徹地指出，在編譯期中，編譯器對於如何產生正確的呼叫動作，可以完全不需任何特定知識。對 **draw()** 的每一次呼叫，都是透過動態繫結達成。

擴充性（Extensibility）

此刻，讓我們回到先前的樂器實例。因爲有了多型，所以你可以依你的需要，將任意數量的新型別加入系統，而無需更動 **tune()**。在妥善設計的 OOP 程式中，大多數或甚至全部的函式都會依循 **tune()** 的模式，並且只與 base class 介面相溝通。此類程式被稱爲可擴充的（extensible），因爲你可以藉由「從共通的 base class 繼承出新的資料型別」來加入新功能。處理 base class 介面的那些函式，完全不需任何更動就可以因應新 classes 的加入。

仔細想想，在樂器實例中，如果將更多函式加至 base class，並加上許多新的 classes，會發生什麼事？下面是一張圖示：



是的，不需更動 **tune()**，所有新 classes 便都能和舊 classes 相安無事。即使 **tune()** 被置於另一個檔案，而 **Instrument** 介面也加入了一些新函式，但 **tune()** 仍然無需重新編譯便能運作無誤。以下便是這個程式的實作內容：

```
//: c07:music3:Music3.java
// An extensible program.
import java.util.*;

class Instrument {        // 樂器
  public void play() {
    System.out.println("Instrument.play()");
  }
  public String what() {
    return "Instrument";
  }
```

```
  public void adjust() {}
}

class Wind extends Instrument {     // 管樂器
  public void play() {
    System.out.println("Wind.play()");
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}

class Percussion extends Instrument { // 敲擊樂器
  public void play() {
    System.out.println("Percussion.play()");
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}

class Stringed extends Instrument {      // 弦樂器
  public void play() {
    System.out.println("Stringed.play()");
  }
  public String what() { return "Stringed"; }
  public void adjust() {}
}

class Brass extends Wind {    // 銅管
  public void play() {
    System.out.println("Brass.play()");
  }
  public void adjust() {
    System.out.println("Brass.adjust()");
  }
}

class Woodwind extends Wind { // 木管
  public void play() {
    System.out.println("Woodwind.play()");
  }
```

```
  public String what() { return "Woodwind"; }
}

public class Music3 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void tune(Instrument i) {
    // ...
    i.play();
  }
  static void tuneAll(Instrument[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
  public static void main(String[] args) {
    Instrument[] orchestra = new Instrument[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();
    tuneAll(orchestra);
  }
} ///:~
```

新增加的函式是 **what()** 和 **adjust()**。**what()** 將 class 描述文字以 **String** reference 回傳，**adjust()** 則提供樂器調音方式。

在 **main()** 中，當你將某個 reference 置於 **Instrument** array 內，形同自動向上轉型至 **Instrument**。

我們發現，**tune()** 很幸福地完全不知道它週遭程式碼所發生的變動，依舊正常運作。這正是我們所期望的多型帶來的效果。你所做的程式碼更動，並不會傷害到程式中不應被影響的部份。換個角度來看，讓程式員「將變動的事物與不變的事物隔離」的所有技術中，多型是最重要的技術之一。

覆寫（overriding）vs. 重載（overloading）

讓我們換個角度看看本章的第一個範例。下列程式中，**play()** 的介面在覆寫過程中被改變了，這表示你其實並沒有覆寫（*overridden*）此一函式，而是加以重載（*overloaded*）。編譯器允許你進行函式的重載（多載化），所以不會提出怨言。但這裡的行爲或許並非你所想要：

```
//: c07:WindError.java
// Accidentally changing the interface.

class NoteX {
  public static final int
    MIDDLE_C = 0, C_SHARP = 1, C_FLAT = 2;
}

class InstrumentX {
  public void play(int NoteX) {
    System.out.println("InstrumentX.play()");
  }
}

class WindX extends InstrumentX {
  // OOPS! Changes the method interface:
  public void play(NoteX n) {
    System.out.println("WindX.play(NoteX n)");
  }
}

public class WindError {
  public static void tune(InstrumentX i) {
    // ...
    i.play(NoteX.MIDDLE_C);
  }
  public static void main(String[] args) {
    WindX flute = new WindX();
    tune(flute); // Not the desired behavior!
  }
} ///:~
```

這個例子還呈現了另一件令人困惑的事實。在 **InstrumentX** 中，**play()** 接受 **int**，並以 **NoteX** 做為識別字。是的，即使 **NoteX** 是個 class 名稱，它仍舊可被用來做為識別字而不會引發任何錯誤訊息。而在 **WindX** 中，**play()** 接收一個 **NoteX** reference 並以 **n** 做為識別字（當然你也可以寫成 **play(NoteX NoteX)** 而不會獲得任何編譯錯誤）。明眼人一下子便可看出，程式員想覆寫 **play()**，卻在撰寫函式時出現小小的打字失誤。請注意，如果你遵守標準的 Java 命名習慣，引數識別字應該是 **noteX**（小寫的 **'n'**），因而可以和 class 名稱有所區別。

tune() 之內部發送 **play()** 訊息給 **InstrumentX i**，並以 **NoteX** 的成員之一（**MIDDLE_C**）做為引數。由於傳入的 **NoteX** 成員是 **int**，所以呼叫的是本例重載後的 **play()** 的 **int** 版本 — 因為沒有 **play()** 覆寫版本可用。

以下是輸出結果：

```
InstrumentX.play()
```

這當然不是一種 *polymorphical method call*。了解事情的原由之後，我們便能輕易修正此一問題。但如果這個問題被埋藏於大型程式中，請你想想，找出這個問題的難度有多高。

Abstract classes（抽象類別）和 Abstract methods（抽象函式）

前述的樂器實例中，base class 內的所有函式都只是掛名性質。如果這些函式被呼叫了，可能會引發一些問題。這是因為 **Instrument** 的存在目的僅止於為它的所有 derived classes 提供「共同的介面」。

建立此一共同介面的唯一理由是，任何子型別都可以以不同方式來表現此一共同介面。共同介面建立了一個基本形式，讓你可以陳述所有 derived classes 的共同點。

另一種說法便是將 **Instrument** 稱爲「抽象基礎類別（*abstract base class*）」，或簡稱「抽象類別（*abstract class*）」。當你想要透過共通介面來操作一組 classes 時，便可撰寫 abstract class。Derived class 中所有與「base class 所宣告之標記式」相符的函式，都會透過動態繫結的機制來呼叫。然而正如前一節所言，如果某個函式名稱和 base class 函式名稱相同，但引數相異，這是一種重載（overloading）行爲，而非覆寫（overriding）行爲。這或許不是你想要的。

如果你撰寫了一個像 **Instrument** 這樣的 abstract class，那麼其物件幾乎沒有任何意義。也就是說 **Instrument** 只被用來表示介面，沒有專屬的實作內容。因此，產生 **Instrument** 物件不僅完全沒有意義，你甚至可能希望阻止使用者這麼做。只要讓 **Instrument** 的所有函式都印出錯誤訊息，就可以達成這個目的，但這麼做得等到執行期才能透露出這項資訊，而且得在客戶端進行可信賴的持久性測試。如果我們能在編譯期就找出問題來，當然最好。

對此，Java 提供了所謂 *abstract method* [1]機制。這是一種不完全的函式，只有宣告而無本體。以下便是 abstract method 的宣告語法：

```
abstract void f();
```

含有 abstract methods（抽象函式）者我們稱爲 *abstract class*（抽象類別）。如果 class 含有單一或多個 abstract methods，便需以關鍵字 **abstract** 做爲這個 class 的飾詞，否則編譯器會發出錯誤訊息。

如果 class 是個半成品，那麼當我們試著產生其物件時，編譯器如何反應呢？唔，爲 abstract class 產生任何物件都是不安全的，編譯器會發出錯誤訊息。這是編譯器確保 abstract class 純粹性的方式，因此你不必擔心自己誤用它。

如果你繼承一個 abstract class，並且希望爲新型別產生物件，那麼你得爲 base class 中的所有 abstract methods 都提供相應的定義。如果沒有這麼做

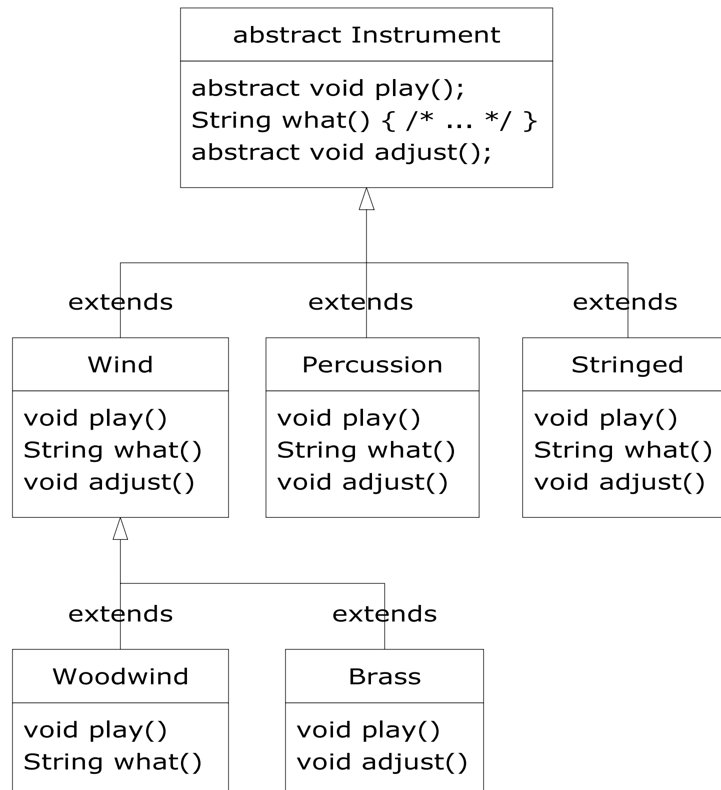
---

[1]對 C++程式員而言，abstract method 相當於 C++ 的純虛擬函式（*pure virtual function*）。

（這是你的選擇），derived class 便也成爲一個 abstract class，而且編譯器會強迫你以關鍵字 **abstract** 來修飾這個 derived class。

我們也可以將不含任何 **abstract** methods 的 class 宣告爲 **abstract**。如果你不希望你所撰寫的 class 被產生出任何實體，但這個 class 又不具備「擁有 **abstract** methods」的實際理由時，這項性質便極爲有用。

Instrument class 可被輕易轉變爲 **abstract** class。這其中只有某些函式會變成抽象，因爲將某個 class 宣告爲 **abstract**，並不強迫你得將所有函式都宣告爲 **abstract**。以下是可能的樣子：



以下便是運用了 **abstract** classes（抽象類別）和 **abstract** methods（抽象函式）之後的管弦樂器修定版：

```java
//: c07:music4:Music4.java
// Abstract classes and methods.
import java.util.*;

abstract class Instrument {
  int i; // storage allocated for each
  public abstract void play();
  public String what() {
    return "Instrument";
  }
  public abstract void adjust();
}

class Wind extends Instrument {
  public void play() {
    System.out.println("Wind.play()");
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}

class Percussion extends Instrument {
  public void play() {
    System.out.println("Percussion.play()");
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}

class Stringed extends Instrument {
  public void play() {
    System.out.println("Stringed.play()");
  }
  public String what() { return "Stringed"; }
  public void adjust() {}
}

class Brass extends Wind {
  public void play() {
    System.out.println("Brass.play()");
  }
```

```
  public void adjust()
    System.out.println("Brass.adjust()");
  }
}

class Woodwind extends Wind {
  public void play() {
    System.out.println("Woodwind.play()");
  }
  public String what() { return "Woodwind"; }
}

public class Music4 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void tune(Instrument i) {
    // ...
    i.play();
  }
  static void tuneAll(Instrument[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
  public static void main(String[] args) {
    Instrument[] orchestra = new Instrument[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();
    tuneAll(orchestra);
  }
} ///:~
```

你看到了，除了 base class，實際上沒有任何改變。

撰寫 **abstract** classes 和 **abstract** methods 是很實用的，因爲它們可以明確宣示 class 的抽象性質，並告訴使用者和編譯器它所設想的被運用方式。

建構式（Constructors）和多型（polymorphism）

一如以往，建構式異於其他函式。即便現在加入了多型，這句話仍然成立。雖然建構式不具多型性格（但你還是可以擁有某種「虛擬建構式」，詳見 12 章），但是了解如何在複雜的繼承架構中以建構式搭配多型，還是相當重要的。這樣的了解可以幫助你避免一些令人不快的困擾。

建構式叫用順序（order of constructor calls）

建構式的叫用順序，第 4 章已經簡短討論過，第 6 章亦再次討論。但那些討論都是在多型被引入之前。

derived class 建構式一定會呼叫 base class 建構式，由此將繼承階層串連起來，使每個 base class 的建構式皆被喚起。這麼做是有用的，因爲建構式有個十分特殊的任務：檢視物件是否被妥善建立。derived class 僅能存取其自身的成員，無法存取 base class 的成員（那些成員通常被宣告爲 **private**）。唯有 base class 建構式才具備合宜的知識，可以將自身元素加以初始化。也唯有 base class 建構式才能存取其自身元素。因此，讓所有建構式都能夠被呼叫到，是十分重要的，否則整個物件便無法建構成功。這就是編譯器之所以「強迫 derived class 建構式必須呼叫 base class 建構式」的原因。如果你未在 derived class 建構式本體中明確呼叫 base class 建構式，base class 的 *default* 建構式便會被自動喚起。如果缺乏 *default* 建構式，編譯器便發出錯誤訊息。（注意：在 class 不帶有建構式的情形下，編譯器會自動合成出一個 *default* 建構式）。

讓我們看看下面這個例子，它說明了複合（composition）、繼承（inheritance）、多型（polymorphism）在建構順序上產生的效應：

```
//: c07:Sandwich.java
// Order of constructor calls.

class Meal {
  Meal() { System.out.println("Meal()"); }
}
```

```
class Bread {
  Bread() { System.out.println("Bread()"); }
}

class Cheese {
  Cheese() { System.out.println("Cheese()"); }
}

class Lettuce {
  Lettuce() { System.out.println("Lettuce()"); }
}

class Lunch extends Meal {
  Lunch() { System.out.println("Lunch()");}
}

class PortableLunch extends Lunch {
  PortableLunch() {
    System.out.println("PortableLunch()");
  }
}

class Sandwich extends PortableLunch {
  Bread b = new Bread();
  Cheese c = new Cheese();
  Lettuce l = new Lettuce();
  Sandwich()
    System.out.println("Sandwich()");
  }
  public static void main(String[] args) {
    new Sandwich();
  }
} ///:~
```

這個例子以其他 classes 爲素材，製作出一個複雜的 class；每個 class 都擁有一個能印出自己名稱的建構式。最重要的 class 是 **Sandwich**，它反映出三階繼承（如果你把內定的 **Object** 繼承也算在內的話，則是四階繼承），並擁有三個成員物件。當 **Sandwich** 物件在 **main()** 中被產生出來，其輸出結果是：

```
Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwich()
```

這說明了此一複雜物件的建構式叫用順序如下：

1. 呼叫 base class 建構式。這個步驟會反覆遞迴，使繼承階層的根源最先被建構，然後是次一層 derived class，直至最末一層 derived class 爲止。

2. 根據各個成員的宣告順序，呼叫各個成員的初值設定式（initializers）。

3. 呼叫 derived class 建構式本體。

建構式的叫用順序很重要。當你動用繼承機制，你已知道 base class 的所有資訊，並可存取 base class 中宣告爲 **public** 和 **protected** 的所有成員。這意味當你置身於 derived class 時，你必須假設 base class 的所有成員都是有效的。在一般函式之內，建構動作早已完成，所以物件內的所有成份（譯註：因複合而形成的東西）的所有成員皆已建構完成。然而在建構式中，你得設法讓你所使用的成員事先建構完成。要得到這樣的保證，方法之一便是讓 base class 的建構式先一步被喚起。然後，當你置身於 derived class 建構式時，base class 中所有可供取用的成員便已初始化完畢。在建構式中「確知所有成員皆可被合法取用」這一理由，促使我們儘可能在成員物件（亦即複合狀態，如上例的 **b**、**c**、**l**）出現於 class 定義式時，便將它們初始化。如果你依循這個習慣，便能夠確保所有 base classes 的成員以及當前這個物件的成員物件都已被初始化。不幸的是，這種做法仍然無法因應所有情況。詳見下節。

繼承與 finalize()

當你使用複合技術來撰寫新的 class，你不需要煩惱它的成員物件的終止（*finalizing*）問題。每個成員都是獨立物件，因此會被垃圾回收器回收並終止。這和它恰巧是你的 class 成員沒有任何關係。不過，在繼承關係中，如果你有某些特別的清理動作（cleanup）得在垃圾回收時進行，你就得在 derived class 中覆寫 **finalize()**。如果你這麼做，千萬別忘了呼叫 base class 的 **finalize()**，因為如果不這樣，base class 的終止動作（*finalization*）就不會發生。下面是一個驗證：

```
//: c07:Frog.java
// Testing finalize with inheritance.

class DoBaseFinalization {
  public static boolean flag = false;
}

class Characteristic {
  String s;
  Characteristic(String c) {
    s = c;
    System.out.println(
      "Creating Characteristic " + s);
  }
  protected void finalize() {
    System.out.println(
      "finalizing Characteristic " + s);
  }
}

class LivingCreature {
  Characteristic p =
    new Characteristic("is alive");
  LivingCreature() {
    System.out.println("LivingCreature()");
  }
  protected void finalize() throws Throwable {
    System.out.println(
      "LivingCreature finalize");
```

```
    // Call base-class version LAST!
    if(DoBaseFinalization.flag)
      super.finalize();
  }
}

class Animal extends LivingCreature {
  Characteristic p =
    new Characteristic("has heart");
  Animal() {
    System.out.println("Animal()");
  }
  protected void finalize() throws Throwable {
    System.out.println("Animal finalize");
    if(DoBaseFinalization.flag)
      super.finalize();
  }
}

class Amphibian extends Animal {
  Characteristic p =
    new Characteristic("can live in water");
  Amphibian() {
    System.out.println("Amphibian()");
  }
  protected void finalize() throws Throwable {
    System.out.println("Amphibian finalize");
    if(DoBaseFinalization.flag)
      super.finalize();
  }
}

public class Frog extends Amphibian {
  Frog() {
    System.out.println("Frog()");
  }
  protected void finalize() throws Throwable {
    System.out.println("Frog finalize");
    if(DoBaseFinalization.flag)
      super.finalize();
  }
```

```
  public static void main(String[] args) {
    if(args.length != 0 &&
       args[0].equals("finalize"))
       DoBaseFinalization.flag = true;
    else
      System.out.println("Not finalizing bases");
    new Frog(); // Instantly becomes garbage
    System.out.println("Bye!");
    // Force finalizers to be called:
    System.gc();
  }
} ///:~
```

class **DoBaseFinalization** 中只有一個旗標，用以代表「每個繼承階層中的 class 是否呼叫 **super.finalize()**」。程式會依據命令列引數（command line argument）來設定旗標值，讓你觀察帶有（或不帶有）base class 終止動作的行爲表現。

在這個階層體系中，每個 class 都含有一個 **Characteristic** 成員物件。你會發現，無論 base class 的終止式（finalizers）是否被呼叫，**Characteristic** 成員物件都一定會被終止。

每個被覆寫的 **finalize()** 至少得具有存取 **protected** 成員的權限，因爲 class **Object** 的 **finalize()** 是個 **protected** 函式，而編譯器不允許你在繼承過程中降低存取權限（注意："Friendly" 的存取權限小於 **protected**）

在 **Frog.main()** 中，**DoBaseFinalization** 旗標會被設立，並產生一個 **Frog** 物件。記住，垃圾回收動作（更明確地說是終止動作）可能不會發生於某個物件身上。所以，爲了強迫讓垃圾回收動作發生，我們呼叫 **System.gc()** 觸發垃圾回收動作的進行，並因此引發終止動作。如果不進行 base class 終止動作，輸出結果是：

```
Not finalizing bases
Creating Characteristic is alive
LivingCreature()
Creating Characteristic has heart
Animal()
Creating Characteristic can live in water
Amphibian()
```

```
Frog()
Bye!
Frog finalize
finalizing Characteristic is alive
finalizing Characteristic has heart
finalizing Characteristic can live in water
```

你可以看到，的確沒有 **Frog** 的任何 base classes 的 finalizers（終止式）被喚起（至於其成員物件則如你所預期地被終止了）。如果你在命令列加入 "finalize" 引數，你會得到這樣的結果：

```
Creating Characteristic is alive
LivingCreature()
Creating Characteristic has heart
Animal()
Creating Characteristic can live in water
Amphibian()
Frog()
bye!
Frog finalize
Amphibian finalize
Animal finalize
LivingCreature finalize
finalizing Characteristic is alive
finalizing Characteristic has heart
finalizing Characteristic can live in water
```

雖然成員物件的終止順序和生成順序相同，但技術上而言，物件的終止順序是未經規範的。不過，透過 base classes，你可以控制終止順序。最佳順序便如此處所示範，也就是與初始化順序恰恰相反。依循此種（C++用於解構式的）形式，你應該先執行 derived class 的終止式，然後才是 base class 的終止式。這是因爲 derived class 的終止動作可能會呼叫某些 base class 函式，而這些終止動作的正常運作有賴其「base class 成份」仍舊可用才行，所以你不能過早加以摧毀。

polymorphic methods 在建構式中的行為

constructor calls（建構式叫用動作）的階層架構，突顯了一個有趣的兩難局面。如果在建構式中呼叫「正在建構中的那個物件」的某個動態繫結的函式，會發生什麼事？如果這發生在一般函式之中，你可以想像會發生什麼事：執行期間會解析這個動態繫結呼叫動作，因爲物件並不知道它究竟屬於此一函式所隸屬的 class，還是屬於其某個 derived class。基於一致性，你可能會認爲這也是建構式中發生的行爲。

事實並非如此。如果你在建構式中呼叫動態繫結的某個函式，會喚起該函式被覆寫（overridden）後的定義。然而其效應無法預期，甚至可能會遮蓋某些難以發現的臭蟲。

就觀念而言，建構式的任務是讓物件從無到有（這很難被視爲一般性工作）。在任何建構式中，整個物件可能僅有部份被形成 － 你只能確知「base class 成份」已形成，但你無法知道有哪些 classes 繼承自你。然而一個動態繫結的 method call 會在繼承階層中向外發散。它會呼叫到 derived class 的函式。如果你在建構式中這麼做的話，你便會喚起某個函式，其中可能取用尚未被初始化的一些成員。這當然是災難的開始。

你可以從以下例子觀察到這個問題：

```
//: c07:PolyConstructors.java
// Constructors and polymorphism
// don't produce what you might expect.

abstract class Glyph {
  abstract void draw();
  Glyph() {
    System.out.println("Glyph() before draw()");
    draw();
    System.out.println("Glyph() after draw()");
  }
}
```

```
class RoundGlyph extends Glyph {
  int radius = 1;
  RoundGlyph(int r) {
    radius = r;
    System.out.println(
      "RoundGlyph.RoundGlyph(), radius = "
      + radius);
  }
  void draw()
    System.out.println(
      "RoundGlyph.draw(), radius = " + radius);
  }
}

public class PolyConstructors {
  public static void main(String[] args) {
    new RoundGlyph(5);
  }
} ///:~
```

在 **Glyph** 中，**draw()** 是抽象的（**abstract**），所以其作用是讓其他人進行覆寫。於是你被強迫在 **RoundGlyph** 中加以覆寫。但 **Glyph** 建構式呼叫了此一函式，結果喚起了 **RoundGlyph.draw()**。這似乎就是我們的意圖，現在請看看輸出結果：

```
Glyph() before draw()
RoundGlyph.draw(), radius = 0
Glyph() after draw()
RoundGlyph.RoundGlyph(), radius = 5
```

當 **Glyph** 建構式呼叫 **draw()** 時，**radius** 尚未被設定初值 1，其值當時為 0。這可能造成只有一個點（或甚至沒有任何東西）被繪製於螢幕上。你會在一旁乾瞪眼，並試圖找出這個程式無法正常運作的原因。

前一節所描述的初始化順序並不十分完整，而這正是此一謎題的解答關鍵。實際的初始化過程是：

1. 任何事情發生之前，配置給此物件的儲存空間會被初始化為二進制零值（binary zero）。
2. 以先前所述方式，呼叫 base class 建構式。此時，覆寫後的 **draw()** 會被呼叫（在 **RoundGlyph** 建構式被呼叫之前）。由於步驟 1 之故，**draw()** 看到的 **radius** 值為零。
3. 以「成員宣告順序」來呼叫各成員的初始式（initializers）。
4. 呼叫 derived class 建構式本體。

這個過程有好的一面，那就是每樣東西至少都會先被初始化為零值（不論零值對特定資料型別而言是否有意義），而不是雜七雜八的內容。即使是「藉由複合手法而被置於 class 內」的 object references，其值都會是 **null**。所以如果你忘了為該 reference 設定初值，便會在執行期收到異常。所有其他物件的值也都是零 — 那常常是檢視輸出結果時顯露問題的地方。

從另一個角度說，這個程式的結果應該會令你感到十分驚恐。你做的一切都符合邏輯，其行為卻不可思議地出錯了，而編譯器沒有給你任何訊息（C++在這個情況下有比較合理的行為）。諸如此類的臭蟲很容易隱匿起來，得花許多時間才能把它們找出來。

因此，撰寫建構式時，一條原則便是：「儘可能簡單地讓物件進入正確狀態。如果可以的話，別呼叫任何函式」。建構式中唯一可以安全呼叫的函式便是「base class 中的 **final** 函式」（這對 **private** 函式來說一樣成立，因為它們天生就是 **final**）。此類函式無法被覆寫，也就不會產生這一類令人驚訝的結果。

將繼承（inheritance）運用於設計

學習了多型（polymorphism）之後，你可能覺得每樣東西都應該被繼承，因為多型是如此巧妙。這麼做可能會造成設計上的負擔；事實上當你使用既有的 class 來產生新的 class 時，如果先選擇繼承手法，有可能導致不必要的複雜情況。

當你不知道該選擇「繼承」或「複合」手法時，最好先選擇複合。複合手法不會強迫你的設計出現一大串繼承階層架構。複合手法的彈性比較大，因爲它可以動態選擇一個型別（也就選擇了其行爲）。如果使用繼承手法，便得在編譯期知道確切型別。以下例子說明這一點：

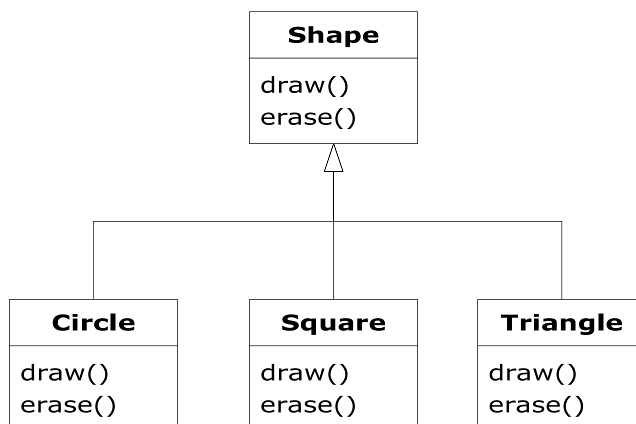
```
//: c07:Transmogrify.java
// Dynamically changing the behavior of
// an object via composition.

abstract class Actor {
  abstract void act();
}

class HappyActor extends Actor {
  public void act()
    System.out.println("HappyActor");
  }
}

class SadActor extends Actor {
  public void act()
    System.out.println("SadActor");
  }
}

class Stage {
  Actor a = new HappyActor();
  void change() { a = new SadActor(); }
  void go() { a.act(); }
}

public class Transmogrify {
  public static void main(String[] args) {
    Stage s = new Stage();
    s.go(); // Prints "HappyActor"
    s.change();
    s.go(); // Prints "SadActor"
  }
} ///:~
```

此例之中，**Stage** 物件含有一個 **Actor** reference，其初值指向一個 **HappyActor** 物件。這表示 **go()** 會產生某種特殊行為。由於 reference 可於執行期被重新繫結至另一個不同的物件，所以我們可以將「指向 **SadActor** 物件」的 reference 置入 **a** 中，造成 **go()** 的行為改變。於是我們獲得了執行期的動態彈性。這種手法又被稱為 *State Pattern*，詳見《*Thinking in Patterns with Java*》（此書可於 *www.BruceEckel.com* 下載）。相較之下，你無法於執行期決定繼承對象，你一定得在編譯期決定之。

下面是個一般準則：「請以繼承表達行為上的差異，以資料成員表達狀態（state）上的變化」。上述例子同時使用了兩者：兩個不同的 classes 被衍生出來，表達 **act()** 的行為差異，**Stage** 則以複合手法來允許狀態變化。本例的狀態變化導致了行為變化。

純粹繼承（Pure inheritance）vs. 擴充（extension）

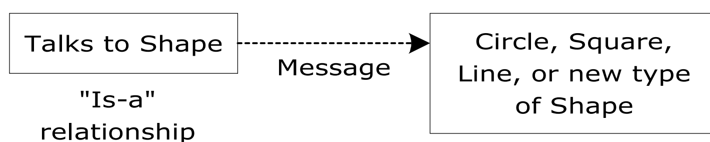
學習繼承時，最聰明的方法似乎是採用「純粹」繼承來建立整個繼承體系。也就是說，只有 base class（或謂 **interface**）所建立的函式，才被 derived class 加以覆寫，如下所示：



這種作法可被視為純粹的 ***is-a***（是一種）關係，因為 class 的介面確立了它究竟是什麼。繼承機制確保所有 derived class 都具備和 base class 相同

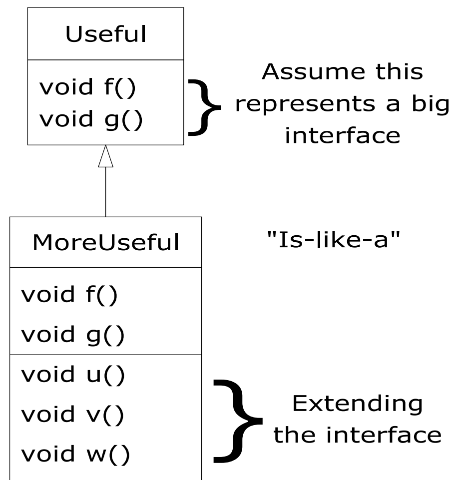
的介面，而且一模一樣。如果你採用上述圖示，那麼 derived classes 除了「base class 介面」之外一無長物。

這種方式可被視爲「純取代（*pure substitution*）」，因爲 derived class 物件可被完全取代爲 base class，而且當你使用它們時，完全不需要知道關於 subclass 的任何額外資訊。

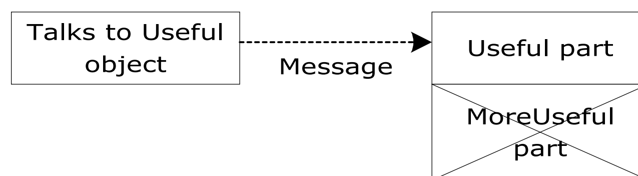


也就是說，base class 可以接收所有發送給 derived class 的訊息，因爲二者具有一模一樣的介面。你只需做一件事，便是將 derived class 向上轉型，完全不需回頭檢視你所處理的物件的確切型別。所有事務都可以透過多型（polymorphism）來處理。

當你從這個角度來理解，似乎只有純粹的 ***is-a*** 關係才是唯一合乎情理的作法。所有其他設計都象徵不潔的思考方式，當然也就是拙劣的設計。是嗎？呃，這是個陷阱。當你深入思考，你馬上會改變想法，並且發現，「擴充介面（extending the interface）」才是解決特定問題的完美解答（關鍵字 **extends** 似乎也在鼓勵大家這麼做）。這種形式可被視爲 ***is-like-a***（像是一個）的關係，因爲 derived class 像是一個 base class：它具備相同的基礎介面，以及由額外函式加以實作的功能。



雖然這是有用而且合理的作法（視運用情況而定），但它仍有缺點。在 derived class 中擴充的介面，在 base class 中無法使用。所以當你向上轉型之後，便無法呼叫新增的函式：

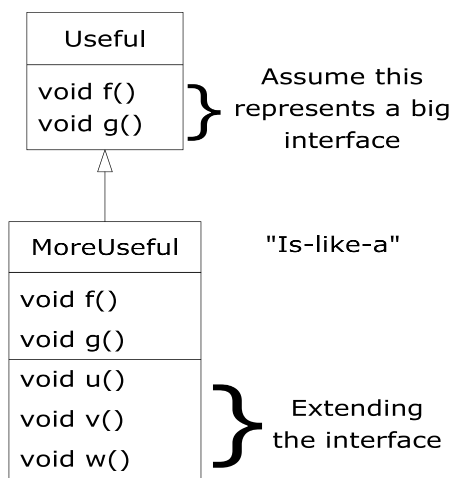


如果你沒有向上轉型，這便不會困擾到你。但你常常會遇到某些情況，使你一定得知道物件的確切型別，才能夠存取該型別所擴充的函式。下一節告訴你該怎麼做。

向下轉型（downcasting）與執行期型別辨識（run-tine type identification）

由於向上轉型（亦即在繼承階層中向上移動）會遺失型別資訊，我們很自然聯想，向下轉型（亦即在繼承階層中向下移動）可以取回型別資訊。不過你知道，向上轉型絕對安全，因爲 base class 不會具備比 derived class

更大型的介面，因此所有經由 base class 介面所發送的訊息，都保證會被接受。但使用向下轉型時，你無法明確知道「某個形狀實際上是圓形」。喔，它可能是三角形、正方形、或其他形狀。



爲了解決上述問題，必須有某種方法保證向下轉型的正確性，使你不致於一不小心做了錯誤的轉型動作，進而送出該物件無法接受的訊息 — 這將是極不安全的動作。

某些程式語言（如 C++）爲了確保向下轉型的安全，要求你執行某個特殊動作（譯註：dynamic_cast）。但 Java 裡頭的每個轉型動作都會被檢查！所以即使看起來不過是以小括號表示的一般轉型動作，執行時期卻會加以檢查以確保它的確是你所認知的型別。如果轉型不成功，你便會收到 **ClassCastException**。「在執行時期檢驗型別」這一動作被稱爲「執行期型別辨識（*run-time type identification*，RTTI）」。下例說明 RTTI 的行爲：

```
//: c07:RTTI.java
// Downcasting & Run-time Type
// Identification (RTTI).
import java.util.*;

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 w() {}
}

public class RTTI {
  public static void main(String[] args) {
    Useful[] x = {
      new Useful(),
      new MoreUseful()
    };
    x[0].f();
    x[1].g();
    // Compile-time: method not found in Useful:
    //! x[1].u();
    ((MoreUseful)x[1]).u(); // Downcast/RTTI
    ((MoreUseful)x[0]).u(); // Exception thrown
  }
} ///:~
```

一如上頁圖片所示，**MoreUseful** 擴充了 **Useful** 的介面。但因爲它是繼承而來，所以它也可以向上轉型至 **Useful**。你可以在 **main()** 的 array **x** 初始化過程中看到這樣的向上轉型動作。由於 array 中的兩個物件都是 class **Useful**，所以你可以對著兩者呼叫 **f()** 和 **g()**。如果你嘗試呼叫 **u()**（它僅存於 **MoreUseful** 內），便會收到編譯期錯誤訊息。

如果你想取用 **MoreUseful** 物件內的擴充介面，可試著向下轉型。如果你的轉型目標是正確的型別，轉型動作便會成功。反之則會收到 **ClassCastException**。你不需要爲此異常撰寫任何程式碼，因爲它所代表的是任何程式員可能在任何程式地點犯下的一種錯誤。

RTTI 比單純的轉型包含了更多內涵。舉例而言，有個方法可以讓你在嘗試向下轉型之前，先取得你所處理的型別。第 12 章便是討論 Java 執行期型別辨識（RTTI）機制的各個不同面向。

摘要

多型（polymorphism）意謂「不同的形式（difference forms）」。在物件導向程式設計中，你會有同一份表徵（face，亦即 base class 所提供的共同介面），以及不同的表徵運用形式：不同版本的動態繫結函式（*dynamically bound methods*）。

你已經在本章中看到，如果不使用資料抽象性和繼承，就不可能了解多型並進而運用多型。多型是一個無法被單獨對待的特性，只能協同運作，作爲「class 相對關係」大局（a "big picture" of class relationshops）中的一部份。人們常會被 Java 的其他「非物件導向功能」混淆，例如「函式重載」有時候會被說成是物件導向性質。啊，千萬別受騙：只要不是後期繫結，就不是多型。

想要有效率地在程式中使用多型（以及物件導向技術），你得延伸你的設計視野，不僅觀察個別 class 的成員和訊息（譯註：此處「訊息」意指 class 能接受的函式呼叫），也要注意 classes 之間的共通性及其彼此關係。雖然這需要付出可觀的心力，但值得。因爲這麼做能夠帶來更快速的程式開發時程、更好的程式組織、可擴充的程式碼、以及更好的維護性。

練習

某些經過挑選的題目，其解答置於《*The Thinking in Java Annotated Solution Guide*》電子文件中。僅需小額費用便可自 *www.BruceEckel.com*.網站取得。

1. 將某個會印出訊息的函式加至 **Shapes.java** 的 base class 內，但不在 derived class 中加以覆寫。請解釋所發生的事情。然後在某個（而非全部）derived class 中覆寫此一函式，並觀察所發生的事情。最後，在所有 derived class 中覆寫此一函式。

2. 將新的 **Shape** 型別加至 **Shapes.java**，並在 **main()** 中檢查多型是否作用於你的新型別身上，就像作用於舊型別那般。

3. 改寫 **Music3.java**，使 **what()** 成爲 **Object** 的 **toString()**。請使用 **System.out.println()**（不進行任何轉型）來印出 **Instrument** 物件內容。

4. 將新的 **Instrument** 型別加入 **Music3.java**，然後檢查多型是否作用於你的新型別身上。

5. 修改 **Music3.java**，使這個程式能以 **Shapes.java** 中的方式隨機產生 **Instrument** 物件。

6. 建立 **Rodent**（齧齒目動物）的繼承階層結構：**Mouse**（家鼠）、**Gerbil**（沙鼠）、**Hamster**（倉鼠）等等。在 base class 中提供所有 **Rodent**s 的共同函式，並於 derived classes 中加以覆寫，根據特定的 **Rodent** 型別，執行不同的行爲。現在，產生一個 **Rodent** array，填入不同的 **Rodent** 型別，並呼叫 base class 函式，觀察究竟會發生什麼事。

7. 修改上題，讓 **Rodent** 成爲 **abstract** class。並將 **Rodent** 的函式宣告爲 **abstract** － 如果可以的話。

8. 產生某個 **abstract** class，並令它不含任何 **abstract** methods。驗證你的確無法爲該 class 產生一份實體。

9. 將 **Pickle**（醃菜）加入 **Sandwich.java**。

10. 修改練習 6，使它展示 base class 和 derived class 的初始化順序。然後在 base class 和 derived classes 中加入成員物件，並顯示建構過程中它們的初始化動作發生順序。

11. 建立一個三階繼承體系。每一階 class 都應該擁有 **finalize()**，而且這些函式都應該適當呼叫 base class 的 **finalize()**。請驗證你的繼承體系運作恰當。

12. 撰寫一個 base class，具有兩個函式，並在第一個函式中呼叫第二個函式。然後，衍生一個新的 class，並於其中覆寫第二個函式。現在，產生一個 derived class 物件，將它向上轉型至基礎型別（base type）並呼叫第一個函式。請解釋所發生的事情。

13. 撰寫一個 base class，具有 **abstract print()**，並在 derived class 中覆寫之。覆寫後的版本會印出 derived class 所定義的某個 **int** 變數值。在此變數的定義地點，給予某個非零值。在 base class 建構式中呼叫此一函式。現在，在 **main()** 中產生一個 derived class 物件，然後呼叫其 **print()**。請解釋所發生的事情。

14. 依循 **Transmogrify.java** 的例子，撰寫一個 **Starship** class，內含一個 **AlertStatus** reference，它可以代表三種不同狀態。加入一些能夠改變狀態（states）的函式。

15. 撰寫一個 **abstract** class，不具任何函式。自此衍生出一個 class，並爲它加上一個函式。撰寫某個 **static** 函式，使其接受一個「指向 base class」的 reference，並將它向下轉型爲 derived class，然後呼叫前述那個函式。請在 **main()** 中驗證這是可行的。接著，將此函式的 **abstract** 飾詞移至 base class，因而免去向下轉型的需要。

8: Interfaces（介面）與 Inner Classes（內隱類別）

Interfaces（介面）和 inner classes（內隱類別），爲你系統中的物件提供更爲精巧的組織與控制方式。

C++ 不含上述兩類機制 — 雖然熟練的 C++ 程式員還是能夠加以模擬。Interfaces 和 inner classes 存在於 Java 之中，反映出它們的重要性已經受到肯定，因而有必要透過語言關鍵字，提供直接的支援。

在第 7 章中，你已經學到了關鍵字 **abstract**。此一關鍵字讓你得以撰寫「在 class 中不具實際定義」的單一或多個函式 — 是的，你只提供介面部份而不提供相應的實作細目。實作細目由繼承者負責撰寫。本章即將介紹的關鍵字 **interface** 則造出完全抽象的 class，絲毫不帶半點實作內容。很快你會學到，**interface** 更勝於 abstract class，因爲我們能夠藉以撰寫出「可被向上轉型爲多個基本型別」的 class，而達到 C++ 多重繼承的變形。

本章的另一個主題 inner classes，乍看之下不過是單純的程式碼隱藏機制：將 classes 置於其他 classes 之中。不過你會學到，inner classes 其實更甚於此，它不僅知曉外圍（surrounding）class 的存在，亦能與之溝通。雖然對大多數人來說，inner classes 是個新觀念，但是以 inner classes 完成的程式碼比較優雅清晰。不過你得花一些時間才能夠自在地以 inner classes 進行設計。

Interfaces（介面）

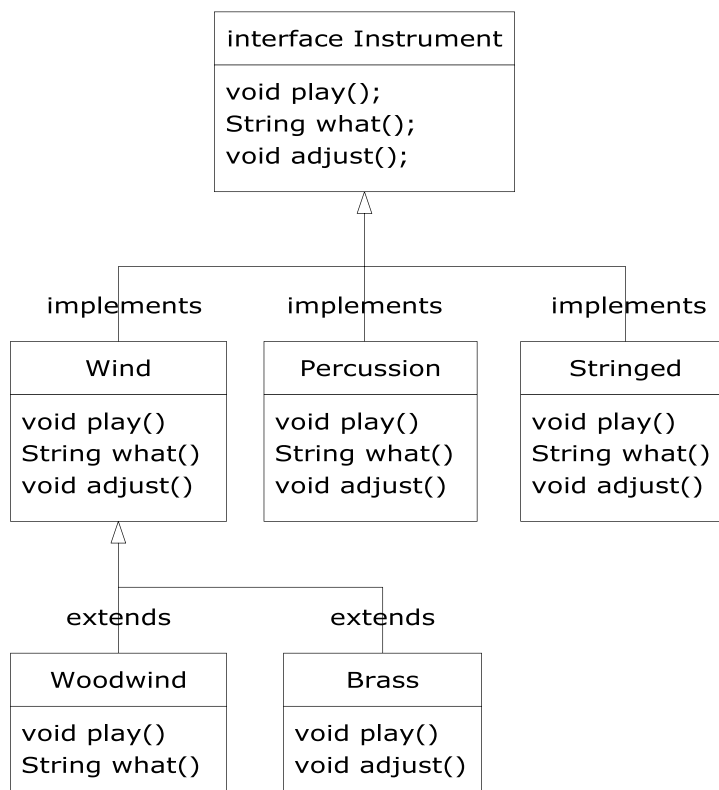
關鍵字 **interface** 將 **abstract** 的概念做了更進一步的發揮。你可以想像它是「純粹」的 **abstract** class。它讓撰寫者得以建構出 class 的形式：函

式的名稱、引數列、回傳型別，但不含函式主體。**interface** 可以內含資料成員，但這些資料成員都會自然而然成為 **static** 和 **final**。是的，**interface** 所提供的只是形式（form），不含實作細目（implementation）。

interface 所陳述的是：「所有實作出本介面的 classes，看起來都應該像這樣」。因此，使用某個 **interface** 的所有程式碼便都知道，經由該 **interface** 所能夠喚起的函式有哪些，而且也就只有那些。可以說，所謂 **interface** 是在 classes 之間建立起一個協定（protocol）。某些物件導向編程語言以關鍵字 *protocol* 來達到相同目的。

撰寫 **interface** 時，請使用關鍵字 **interface** 而非關鍵字 **class**。你可以在關鍵字 **interface** 之前加上關鍵字 **public**（但只有當這個 **interface** 被定義於同名檔案中才行），否則它就成為 "friendly"，僅可被使用於同一個 package 之內。

如果想讓某個 class 符合某一個（或某一組）特定的 **interface**，請使用關鍵字 **implements**。這麼做相當於宣告「所謂 **interface** 只是外觀描述，而我現在要說明它的運作方式」。這種型式看起來像繼承，以下的樂器圖說明了這一點（譯註：請和 p327 圖比較）：



一旦你實作了某個 **interface**，該份實作品便成為一個一般的 class，能以一般方式加以繼承。

你可以將 **interface** 中的函式明白宣告為 **public**。但即便沒有這麼宣告，它們也會是 **public**。所以當你實作（implement）某個 **interface** 時，必須將承襲自該 **interface** 的所有函式都定義為 **public**。否則它們的預設屬性會是 "friendly"，而你也就因此在繼承過程中降低了存取權限 － 這在 Java 編譯器中是不允許的。

從新版的 **Instrument** 中你可以觀察到這一點。請注意，**interface** 中的每個函式都只能是宣告，因為編譯器也只允許你這麼做。此外，**Instrument** 中的眾多函式，沒有任何一個被宣告為 **public**，但它們仍將自動地被設為 **public**：

```
//: c08:music5:Music5.java
// Interfaces.
import java.util.*;

interface Instrument {
  // Compile-time constant:
  int i = 5; // static & final
  // Cannot have method definitions:
  void play(); // Automatically public
  String what();
  void adjust();
}

class Wind implements Instrument {
  public void play() {
    System.out.println("Wind.play()");
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}

class Percussion implements Instrument {
  public void play() {
    System.out.println("Percussion.play()");
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}

class Stringed implements Instrument {
  public void play() {
    System.out.println("Stringed.play()");
  }
  public String what() { return "Stringed"; }
  public void adjust() {}
}

class Brass extends Wind {
  public void play() {
    System.out.println("Brass.play()");
  }
```

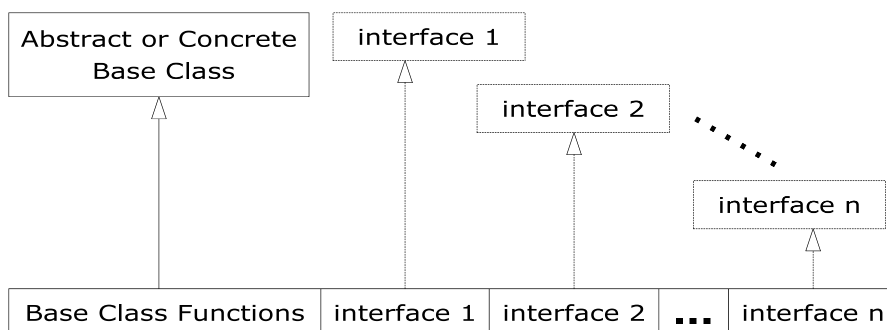
```
  public void adjust() 
    System.out.println("Brass.adjust()");
  }
}

class Woodwind extends Wind {
  public void play() {
    System.out.println("Woodwind.play()");
  }
  public String what() { return "Woodwind"; }
}

public class Music5 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void tune(Instrument i) {
    // ...
    i.play();
  }
  static void tuneAll(Instrument[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
  public static void main(String[] args) {
    Instrument[] orchestra = new Instrument[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();
    tuneAll(orchestra);
  }
} ///:~
```

其餘程式碼的運作方式不變，它們和「向上轉型的目標究竟是名爲 **Instrument** 的一般 class，抑或名爲 **Instrument** 的 **abstract** class，抑或名爲 **Instrument** 的 **interface**」沒有絲毫關係，行爲都相同。事實上，你可以在 **tune()** 中觀察到，沒有任何跡象顯示 **Instrument** 究竟是

個一般 class 或是個 **abstract** class，或是一個 **interface**。這正是目的所在：每種方法都賦予程式員面對「物件的生成和使用」時有不同的掌控。

Java 的多重繼承（multiple inheritance in Java）

interface 不單只是一種「更純粹」的 **abstract** class，其用途更甚於此。由於 **interface** 不帶任何實作細目，也就是說 **interface** 不與任何儲存空間有關連，因此合併多個 **interfaces** 是一件輕而易舉的工作，編譯器不需如臨大敵。此事極具價值，因爲有時候你需要宣告「**x** 是個 **a**，也是個 **b**，同時也是個 **c**」。在 C++中，「合併多個 class 介面」的行爲稱爲多重繼承（multiple inheritance），其中揹負著某種頗爲棘手的包袱，因爲每個 class 都內含實作細目。在 Java 裡頭你可以進行同樣的動作，但其中只有一個 class 可以擁有實作內容。因此，合併多個介面時，C++所面臨的問題不會出現於 Java 之中：



derived class 並不一定非得有個抽象或具象（無任何 **abstract** methods）的 base class。但如果它繼承自 non-**interface**（譯註：亦即抽象或具象 class），便僅能繼承一個，其餘繼承來源都得是 **interfaces** 才行。請將繼承的 **interfaces** 名稱置於關鍵字 **implements** 之後，並以逗號隔開。繼承而來的 **interfaces** 數目不受限制，每一個都是獨立型別，都可以成爲你向上轉型的目標。以下示範組合多個 **interfaces** 及一個具象 class：

```
//: c08:Adventure.java
// Multiple interfaces.
import java.util.*;
```

```
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 {
  static void t(CanFight x) { x.fight(); }
  static void u(CanSwim x) { x.swim(); }
  static void v(CanFly x) { x.fly(); }
  static void w(ActionCharacter x) { x.fight(); }
  public static void main(String[] args) {
    Hero h = new Hero();
    t(h); // Treat it as a CanFight
    u(h); // Treat it as a CanSwim
    v(h); // Treat it as a CanFly
    w(h); // Treat it as an ActionCharacter
  }
} ///:~
```

你看到了，**Hero** 結合了具象類別 **ActionCharacter**，以及 **CanFight**、**CanSwim**、**CanFly** 等 interfaces。當你經由這種方式，將某個具象類別和其他 interfaces 結合在一起時，你得先寫下具象類別的名稱，然後才是 interfaces 的名稱，否則編譯器會發出錯誤訊息。

請注意，**interface CanFight** 及 class **ActionCharacter** 中的 **fight()** 的標記式（signature）是一致的，而且 **Hero** 之中並未提供 **fight()** 的定義。**interface** 的遊戲規則是，你可以繼承 **interface**（稍後便示範），得到的將又是一個 **interface**。如果你要以此新型別生成一個物件，此一型別必須是個定義齊全的 class。不過雖然 **Hero** 自身並未提供 **fight()** 的定義，但因 **ActionCharacter** 擁有 **fight()** 的定義，被自動繼承下來，因此我們還是可以產生 **Hero** 物件，沒有問題。

在 class **Adventure** 中你可以觀察到，共有四個函式分別接受不同的 interfaces 和 concrete class 做爲引數。一旦 **Hero** 物件被生成，便可被傳入這些函式中的任何一個。這其實就是將該物件向上轉型至各個對應的 **interface**。Java 對 interface 的設計方式，使得上述動作一點都不麻煩，而且不需要程式員付出額外心力。

請千萬記得，interfaces 存在的根本原因已在上例中展現：能夠被向上轉型至多個基本型別。至於使用 interfaces 的第二個理由，和使用 **abstract** base class（抽象基礎類別）一致：讓客端程式員無法產生其物件，並因此確保這只是一個「介面」（而無實體）。這同時引發了一個問題：究竟應該使用 **interface** 還是使用 **abstract** class？**interface** 同時賦予你 **abstract** class 和 **interface** 的好處，因此如果你的 base class 可以不帶任何函式定義式或任何成員變數，你應該優先選用 **interfaces**。事實上如果你知道某個東西將會成爲 base class，你的優先考量便是使它成爲 **interface**，只有在必須帶有函式定義式或成員變數時，才請改用 **abstract** class，或甚至（必要時）改用 concrete class。

介面合併時的名稱衝突（name collisions）問題

當你實作多重介面時，可能會掉進一個小小的陷阱。上例之中 **CanFight** 和 **ActionCharacter** 具有完全相同的 **void fight()**。這沒有問題，因爲兩者的 **fight()** 完全相同。但如果不是這樣呢？以下便是一例：

```
//: c08:InterfaceCollision.java
```

```
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; } // overloaded
}

class C3 extends C implements I2 {
  public int f(int i) { return 1; } // overloaded
}

class C4 extends C implements I3 {
  // Identical, no problem:
  public int f() { return 1; }
}

// Methods differ only by return type:
//! class C5 extends C implements I1 {}
//! interface I4 extends I1, I3 {} ///:~
```

這個例子顯現出困難所在，因爲覆寫（overriding）、實作（implementation）、重載（overloading）通通令人不愉快地混雜在一起，而且重載函式無法僅靠回傳型別（return type）做爲區隔。當最後兩行註解被移走，錯誤訊息便解釋了一切：

```
InterfaceCollision.java:23: f() in C cannot
implement f() in I1; attempting to use
incompatible return type
found   : int
required: void
InterfaceCollision.java:24: interfaces I3 and I1 are
incompatible; both define f
(), but with different return type
```

這裡我刻意結合不同的 interfaces 並使用其中同名的函式，造成程式碼可讀性的混亂。是的，請盡力避免此種情況。

透過繼承來擴充 interface

利用繼承，你可以輕易將新的函式加至 **interface** 中，也可以透過繼承將多個 **interfaces** 結合為一個新的 **interface**。以下便示範這兩種產生新 **interface** 的手法：

```
//: c08:HorrorShow.java
// Extending an interface with inheritance.

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 HorrorShow {
  static void u(Monster b) { b.menace(); }
  static void v(DangerousMonster d) {
    d.menace();
    d.destroy();
  }
  public static void main(String[] args) {
```

```
    DragonZilla if2 = new DragonZilla();
    u(if2);
    v(if2);
  }
} ///:~
```

在這裡，**DangerousMonster** 單純只是擴充 **Monster** 而得到的一個新的 **interface**，實作於 **DragonZilla**。

Vampire 所使用的語法只適用於 interfaces 繼承。一般情況下你只能將 **extends** 應用於單一 class，但由於 **interface** 可由多個 interfaces 組成，所以製作新的 **interface** 時，**extends** 也可用來指涉多個 base interfaces。如你所見，只要以逗號隔開多個 **interface** 名稱即可。

產生常數群（grouping constants）

由於 **interface** 中的所有資料成員都會自動成為 **static** 和 **final**，所以對於常數群的產生（就如 C 或 C++中的 **enum** 一樣），interface 是個十分便利的工具。例如：

```
//: c08:Months.java
// Using interfaces to create groups of constants.
package c08;

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;
} ///:~
```

請注意，具備常數初值的 **static finals**，在 Java 中一律以大寫字母表示，並以底線區隔同一個識別名稱中的不同字詞。

interface 中的資料成員會自動成為 **public**，所以無需明白標示。

現在，你可以透過 package 的運用方式匯入 **c08.*** 或 **c08.Months**，藉以使用目前 package 之外的常數，也可以經由 **Months.JANUARY** 之類的

算式來取用其值。當然，你所取用的僅僅只是 **int**，並不存在 C++ **enum** 所具備的型別安全保證。儘管如此，此一廣泛被運用的技巧仍然可以改善你的程式，避免直接將數字寫死於程式之中 — 那會成爲所謂的「魔術數字（magic numbers）」，使程式碼極難維護。

如果你要額外進行型別安全檢驗，可以設計如下的 class[1]：

```
//: c08:Month2.java
// A more robust enumeration system.
package c08;

public final class Month2 {
  private String name;
  private Month2(String nm) { name = nm; }
  public String toString() { return name; }
  public final static Month2
    JAN = new Month2("January"),
    FEB = new Month2("February"),
    MAR = new Month2("March"),
    APR = new Month2("April"),
    MAY = new Month2("May"),
    JUN = new Month2("June"),
    JUL = new Month2("July"),
    AUG = new Month2("August"),
    SEP = new Month2("September"),
    OCT = new Month2("October"),
    NOV = new Month2("November"),
    DEC = new Month2("December");
  public final static Month2[] month =  {
    JAN, JAN, FEB, MAR, APR, MAY, JUN,
    JUL, AUG, SEP, OCT, NOV, DEC
  };
  public static void main(String[] args) {
    Month2 m = Month2.JAN;
    System.out.println(m);
    m = Month2.month[12];
```

---

[1] 這個作法是受到一封來自 Rich Hoffarth 的電子郵件的啓發。

```
    System.out.println(m);
    System.out.println(m == Month2.DEC);
    System.out.println(m.equals(Month2.DEC));
  }
} ///:~
```

此一 class 名爲 **Month2**，避免和 Java 標準程式庫的 **Month** class 衝突。它是一個具有 **private** 建構式的 **final** class，所以不僅無法被繼承，也無法產生其實體（instances）。這個 class 的所有實體都是它自身的 **final static** 實體：**JAN**、**FEB**、**MAR** 等等。這些物件被用於 array **month** 之中，讓你得以透過數字（而非名稱）來選擇月份。請注意，array **month** 之中多放進了一個 **JAN**，目的是要使其索引偏移一個位置，這麼一來 December 便真的是 12 月。你可以觀察到 **main()** 之中的型別安全性：**m** 是個 **Month2** 物件，所以 **m** 僅可被賦予一個 **Month2** 值。這和前例不同，前例的 **Months.java** 僅提供 **int** 值，因此用來表示月份的 **int** 變數實際上可能被賦予任意整數值，這就不夠安全。

在這種作法之下，== 或 **equals()** 都可被用來進行相等測試，一如 **main()** 最後所示範。

將 interfaces 內的資料成員初始化

定義於 interfaces 內的資料成員會自動成爲 **static** 和 **final**。它們不能是 blank finals（譯註：見 p297），但可以被非常數（non-constant）運算式初始化，例如：

```
//: c08:RandVals.java
// Initializing interface fields with
// non-constant initializers.
import java.util.*;

public interface RandVals {
  int rint = (int)(Math.random() * 10);
  long rlong = (long)(Math.random() * 10);
  float rfloat = (float)(Math.random() * 10);
  double rdouble = Math.random() * 10;
} ///:~
```

由於這些資料成員都是 **static**，所以當 class 首次被載入（此事發生於其中任何一個資料成員首次被取用時），它們便都會被初始化。以下是個簡單測試：

```
//: c08:TestRandVals.java

public class TestRandVals {
  public static void main(String[] args) {
    System.out.println(RandVals.rint);
    System.out.println(RandVals.rlong);
    System.out.println(RandVals.rfloat);
    System.out.println(RandVals.rdouble);
  }
} ///:~
```

當然，這些資料成員並非 interface 的一部份，只不過是被儲存於 interface 的 **static** 儲存區域中罷了。

巢狀的（nesting）interfaces

[2]interfaces 可巢狀位於某個 class 或其他 interface 內。這能發展出許多有趣的性質來：

```
//: c08:NestingInterfaces.java

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();
```

[2] 感謝 Martin Danner 在研討班上提出這個問題。

```
  }
  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();
  }
  // Redundant "public":
  public interface H {
    void f();
  }
  void g();
  // Cannot be private within an interface:
  //! private interface I {}
}

public class NestingInterfaces {
  public class BImp implements A.B {
    public void f() {}
  }
```

```java
  class CImp implements A.C {
    public void f() {}
  }
  // Cannot implement a private interface except
  // within that interface's defining class:
  //! class DImp implements A.D {
  //!  public void f() {}
  //! }
  class EImp implements E {
    public void g() {}
  }
  class EGImp implements E.G {
    public void f() {}
  }
  class EImp2 implements E {
    public void g() {}
    class EG implements E.G {
      public void f() {}
    }
  }
  public static void main(String[] args) {
    A a = new A();
    // Can't access A.D:
    //! A.D ad = a.getD();
    // Doesn't return anything but A.D:
    //! A.DImp2 di2 = a.getD();
    // Cannot access a member of the interface:
    //! a.getD().f();
    // Only another A can do anything with getD():
    A a2 = new A();
    a2.receiveD(a.getD());
  }
} ///:~
```

將某個 interface 巢狀置於某個 class 之中，語法相當直覺。巢狀 interfaces 和非巢狀 interfaces 一樣，也可以擁有 **public** 或 "friendly" 兩種可視性（visibility）。你可以看到，不論 **public** 或 "friendly" 的巢狀介面都可被實作爲 **public**、"friendly"、**private** 三種巢狀類別。

interfaces 也可以是 **private**，就如本例的 **A.D**。這種語法既適用於巢狀 interfaces 也適用於巢狀 classes。巢狀的 **private** interfaces 究竟可以帶來什麼好處呢？你可能會猜想，**private** interface 只可被實作爲巢狀的 **private** class，就像 **DImp** 那樣。但是 **A.DImp2** 的存在卻告訴我們，**private** interface 也能被實作爲一個 **public** class。不過即便如此，**A.DImp2** 也不能爲外界所用。你無法對外界說它實作出一個 **private** interface。因此所謂將 **private** interface 實作出來，只是一種方法，用以強迫「定義於該 interface 中的函式」不要被加上任何型別資訊。這也就是說，不允許任何向上轉型發生。

getD() 引出一個和 **private** interface 相關的問題：這是一個 **public** 函式，回傳一個 reference 指向 **private** interface。你能夠使用這個回傳值來做些什麼事呢？在 **main()** 裡頭你會發現，嘗試使用該回傳值（假設爲 x）的動作失敗了。只有在其回傳值被指派給某個物件，而該物件擁有 x 的使用權，才會成功。這樣的物件在本例是以另一個 **A** 透過 **receiveD()** 取得。

interface **E** 告訴我們，interfaces 可彼此相互套疊。但是套用於 interfaces 身上的種種規則，尤其是「所有 interface 的元素都必須爲 **public**」這一條，在此會被嚴格執行。所以「巢狀位於另一個 interface 內」的那些個 interfaces 會自動成爲 **public**，而且無法被宣告爲 **private**。

NestingInterface 示範了巢狀 interfaces 的多種實作方式。請特別注意，當你實作某個 interface，你無需實作其中任何巢狀 interfaces。此外，**private** interfaces 無法在其所定義的 classes 之外被實作。

或許加入這些特性的最初理由，只是爲了語法上的一致性。但我總認爲，你知道語言的某個特性之後，往往能夠找到這個特性的用處。

Inner classes（內隱類別）

將某個 class 的定義置於另一個 class 定義之中是可行的，此即所謂 *inner class*（內隱類別）。inner class 是個極實用的特性，讓你得以將邏輯相關的 classes 組織起來，並讓你得以控制「某個 class 在另一個 class 中的可視

性（visibility）」。不過你一定得知道一件事：inner classes 和所謂複合（composition）是截然不同的兩回事。

通常，當你學習 inner classes 時，需求並不會立刻湧現。本節結束前，當 inner classes 的所有語法和語意都已說明完畢後，你會讀到一些範例，它們應該能夠將 inner classes 帶來的好處解釋清楚。

inner class 的產生方式，就如你所預期的一樣：將 class 的定義置於某個外圍（outer）class 之內：

```
//: c08:Parcel1.java
// Creating inner classes.

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; }
  }
  // Using inner classes looks just like
  // using any other class, within Parcel1:
  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("Tanzania");
  }
} ///:~
```

在 **ship()** 之內使用 inner classes，看起來就和使用其他 classes 一樣。這裡唯一的差別是，其名稱乃巢狀位於 **Parcel1** 之內。稍後你會發現，這其實並非唯一差別。

典型的作法是，外圍 class 有一個函式，可以傳回一個 reference 指向 inner class，像這樣：

```
//: c08:Parcel2.java
// Returning a reference to an inner class.

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 cont()
    return new Contents();
  }
  public void ship(String dest) {
    Contents c = cont();
    Destination d = to(dest);
    System.out.println(d.readLabel());
  }
  public static void main(String[] args) {
    Parcel2 p = new Parcel2();
    p.ship("Tanzania");
    Parcel2 q = new Parcel2();
    // Defining references to inner classes:
    Parcel2.Contents c = q.cont();
    Parcel2.Destination d = q.to("Borneo");
  }
} ///:~
```

如果你想要在外圍 class 的 non-**static** 函式之外產生一個 inner class 物件，你就得以 *OuterClassName.InnerClassName* 的型式指定該物件的型別，一如上述 **main()** 的行為。

Inner classes（內隱類別）與 upcasting（向上轉型）

截至目前，inner classes 似乎並不多麼引人注目。畢竟，如果只是為了隱藏，Java 具備了極佳的隱藏機制 － 讓 class 成為 "friendly"（於是只能在同一個 package 內被看到），不必將它製作成 inner class。

但是當你開始向上轉型至 base class，尤其是轉型為 **interface**，就能突顯 inner classes 的好處（注意，從某個「實作出 interface I」的 inner class 物件身上產生一個 reference 指向 I，本質上和「向上轉型至 base class」是一樣的），這是因為 inner class（也就是 I 的實現者）可以在接下來的情境中完全不被看見，而且不為任何人所用。這麼一來我們就很方便能夠「隱藏實作細目」。你所得到的只是「指向 base class 或 **interface**」的一個 reference 而已。

首先，共有的 interfaces 會被定義於專屬的檔案中，這使它們得以被用於所有範例之中：

```
//: c08:Destination.java
public interface Destination {
  String readLabel();
} ///:~
```

```
//: c08:Contents.java
public interface Contents {
  int value();
} ///:~
```

現在，**Contents** 和 **Destination** 表現出客端程式員可使用的 interfaces。記住，**interface** 會自動將其所有成員都設為 **public**。

當你得到一個 reference，指向 base class 或 base **interface** 時，你甚至可能無法找到其確切型別，一如下例所示：

```
//: c08:Parcel3.java
// Returning a reference to an inner class.
```

```java
public class Parcel3 {    // 譯註：以下 Contents 見 p368
  private class PContents implements Contents
    private int i = 11;
    public int value() { return i; }
  }
  protected class PDestination
      implements Destination { // 譯註：Destination 見 p368
    private String label;
    private PDestination(String whereTo) {
      label = whereTo;
    }
    public String readLabel() { return label; }
  }
  public Destination dest(String s) {
    return new PDestination(s);
  }
  public Contents cont()
    return new PContents();
  }
}

class Test {
  public static void main(String[] args) {
    Parcel3 p = new Parcel3();
    Contents c = p.cont();
    Destination d = p.dest("Tanzania");
    // Illegal -- can't access private class:
    //! Parcel3.PContents pc = p.new PContents();
  }
} ///:~
```

請注意，由於 **main()** 位於 **Test** 之中，所以當你想要執行這個程式時，你不該執行 **Parcel3**，而是應該執行：

```
java Test
```

在這個例子中，**main()** 必須位於另一個獨立 class 內，才能夠說明 inner class **PContents** 的隱密性。

Parcel3 之內有了一些新東西：由於 inner class **PContents** 是為 **private**，所以除了 **Parcel3** 之外，其他 classes 都無法存取之。**PDestination** 是為 **protected**，所以除了 **Parcel3**、**Parcel3** package 內的其他 classes、以及 **Parcel3** 的繼承者之外，其他 classes 皆無法存取 **PDestination**。上述第二個原因是 **protected** 也能帶來「package 存取權限」，也就是說 **protected** 也包括了 "friendly"。以上意味客端程式員不僅所知受限，對這些成員的存取能力也同樣受限。事實上你甚至無法向下轉型至一個 **private** inner class（或一個 **protected** inner class，除非你是其繼承者），因為你根本無法取用其名稱，一如 class **Test** 中所見。因此，**private** inner class 讓 class 設計者得以完全避免任何「與型別相依的程式碼」，並得以完全隱藏實作相關的種種細節。此外，從客端程式員的角度來看，interface 的擴充是沒有用的，因為對客端程式員而言，所有不在 **public interface** class 內的函式都無法取用。這使得 Java 編譯器有機會產生更高效率的碼。

一般的（non-inner）classes 無法被宣告為 **private** 或 **protected**，只能是 **public** 或 "friendly"。

位於 methods 和 scopes 之內的 inner classes（內隱類別）

截至目前你所見到的內容完全圍繞在 inner classes 一般用途上。通常你所撰寫並閱讀的 inner classes 程式碼，都是單純率直易於理解的 inner classes。然而 inner classes 的設計極為完整，如果你需要，它還提供一些隱晦的使用方式：你可以將 inner classes 置於函式之內或甚至置於任意程式範疇（scopes）之內。基於兩個理由你可能會想那麼做：

1. 一如先前所見，你想要實作某種 interface，使你得以產生並回傳某個 reference。

2. 你正在解決某個複雜問題，而你希望在解決方案中設計某個 class，又不希望這個 class 被外界所用。

以下例子會修改先前出現的程式碼，以便運用下面各種東西：

1. 定義於函式之內的 class。

2. 定義於函式內某一段範疇（scope）內的 class。

3. 一個匿名（anonymous）class，用以實作某個 interface。

4. 一個匿名 class，用來擴充一個擁有 non-*default* 建構式的 class。

5. 一個匿名 class，用來執行資料成員初始化動作。

6. 一個匿名 class，以實體（instance）初始化來執行建構動作。注意，匿名的 inner classes 不得擁有建構式。

雖然 **Wrapping** 只是個一般的（帶有實作細目的）class，但它同時被用來做爲其 derived classes 的共同介面。

```
//: c08:Wrapping.java
public class Wrapping {
  private int i;
  public Wrapping(int x) { i = x; }
  public int value() { return i; }
} ///:~
```

請注意，上述 **Wrapping** 的建構式要求傳入一個引數，這使得事情稍爲有趣些。第一個例子示範如何在函式的某個範疇內（而非另一個 class 範疇內）製作 inner class：

```
//: c08:Parcel4.java
// Nesting a class within a method.

public class Parcel4 {
  public Destination dest(String s) {
    class PDestination    // 譯註：以下 Destination 見 p368
        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) {
    Parcel4 p = new Parcel4();
    Destination d = p.dest("Tanzania");
  }
} ///:~
```

class **PDestination** 是 **dest()** 的一部份，不是 **Parcel4** 的一部份（請注意，對同一磁碟目錄下的每個 class 內的 inner class 而言，你可以使用相同 class 名稱而不至於發生衝突），因此你無法在 **dest()** 之外取用 **PDestination**。注意發生於 **return** 述句中的向上轉型動作 – **dest()** 只回傳一個 reference 指向 base class **Destination**。當然啦，雖然 **dest()** 內的 class 是 **PDestination**，但這並不意味 **PDestination** 物件在 **dest()** 回返之後就成了一個不合法的物件。

以下例子示範如何將 inner class 巢狀置於任意程式範疇（scope）內：

```
//: c08:Parcel5.java
// Nesting a class within a scope.

public class Parcel5 {
  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();
    }
    // Can't use it here! Out of scope:
    //! TrackingSlip ts = new TrackingSlip("x");
  }
  public void track() { internalTracking(true); }
  public static void main(String[] args) {
    Parcel5 p = new Parcel5();
    p.track();
  }
```

```
} ///:~
```

class **TrackingSlip** 被巢狀置於 **if** 述句所形成的範疇內。這並不意味該 class 會隨條件的成立才被產生，啊不，它會和其他 classes 一起被編譯出來。不過它在它所處的範疇之外就無法被使用。除此之外看起來和一般 class 沒什麼兩樣。

匿名的（anonymous）inner classes

以下例子看起來有點奇怪：

```
//: c08:Parcel6.java
// A method that returns an anonymous inner class.

public class Parcel6 {
  public Contents cont() {
    return new Contents() {
      private int i = 11;
      public int value() { return i; }
    }; // Semicolon required in this case
  }
  public static void main(String[] args) {
    Parcel6 p = new Parcel6();
    Contents c = p.cont();
  }
} ///:~
```

cont() 竟然將「回傳值的產生」和「用以表現回傳值」的那個 class 的定義合併在一起！而且那個 class 沒有命名 – 是的，它沒有名稱。整個動作看起來像是要產生 **Contents** 物件似的：

```
return new Contents()
```

但是然後（在你看到最後那個分號之前）你說：『等一下，我想我進入了一個 class 定義式』：

```
return new Contents() {
  private int i = 11;
  public int value() { return i; }
};
```

這個奇怪語法的意思是說：「產生某個匿名 class 的物件，此一匿名 class 係繼承自 **Contents**。」**new** 傳回的 reference 會被自動向上轉型爲一個 **Contents** reference。上述匿名 inner class 的語法可被展開爲：

```
class MyContents implements Contents {
  private int i = 11;
  public int value() { return i; }
}
return new MyContents();
```

在此匿名的 inner class 之中，**Contents** 是透過 *default* 建構式產生的。下面這份程式碼示範「當 base class 需要一個帶有引數的建構式時」你該如何處理：

```
//: c08:Parcel7.java
// An anonymous inner class that calls
// the base-class constructor.

public class Parcel7 {
  public Wrapping wrap(int x) {
    // Base constructor call:
    return new Wrapping(x)
      public int value() {
        return super.value() * 47;
      }
    }; // Semicolon required
  }
  public static void main(String[] args) {
    Parcel7 p = new Parcel7();
    Wrapping w = p.wrap(10);
  }
} ///:~
```

是的，只要將適當引數傳入 base class 建構式中，就像將 **x** 傳入 **new Wrapping(x)** 中一樣。匿名 class 不能擁有建構式，所以通常你會呼叫 **super()**。

前兩個例子並不以分號做為 class 主體部份的結束標記（C++則是如此）。取而代之的是，分號用來標示「含有匿名 class」之算式的結束。因此它就像其他地方對分號的運用完全一樣。

如果你得為某個匿名 inner class 的物件執行某種初始化動作，又該如何呢？由於此類 class 不具名稱，因而無法給予其建構式任何名稱 — 因此它無法擁有建構式。不過你還是可以在資料成員定義處執行初始化動作：

```
//: c08:Parcel8.java
// An anonymous inner class that performs
// initialization. A briefer version
// of Parcel5.java.

public class Parcel8 {
  // Argument must be final to use inside
  // anonymous inner class:
  public Destination dest(final String dest) {
    return new Destination() {
      private String label = dest;
      public String readLabel() { return label; }
    };
  }
  public static void main(String[] args) {
    Parcel8 p = new Parcel8();
    Destination d = p.dest("Tanzania");
  }
} ///:~
```

如果你定義了某個匿名 inner class，並且希望用到定義於匿名 inner class 之外的某個物件，編譯器會限制該外部物件必須為 **final**。這也就是為什麼 **dest()** 的引數是 **final** 的原因。如果你忘了這麼做，你會收到編譯期錯誤訊息。

如果你只是簡單地對某個資料成員賦值，上述方式不會有任何問題。但如果你必須執行某些類似建構式的動作，該怎麼辦？透過所謂實體初始化（*instance initialization*）你可以實際完成一個匿名 inner class 的建構：

```
//: c08:Parcel9.java
```

```
// Using "instance initialization" to perform
// construction on an anonymous inner class.

public class Parcel9 {
  public Destination
  dest(final String dest, final float price) {
    return new Destination() {
      private int cost;
      // Instance initialization for each object:
      {
        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) {
    Parcel9 p = new Parcel9();
    Destination d = p.dest("Tanzania", 101.395F);
  }
} ///:~
```

在實體初值設定式（initializer）中你可以看到，有一段程式碼無法做爲資料成員初值設定式的一部份（我說的是那個 **if** 述句）。所以實體初值設定式實際上等於無名 inner class 的建構式。當然啦，其能力受到限制；你無法重載實體初值設定式，所以你只能擁有唯一一個建構式。

與外圍（outer）class 的連結關係

截至目前，inner classes 看起來似乎只不過是一種用於名稱隱藏和程式碼組織的體制。這兩個用途的確有用，但並不能完全讓人信服。是的，其實它還有另一種作用。當你建立一個 inner class 時，其物件便擁有了與其製造者 — 那個外圍（enclosing）物件 — 之間的一種連結關係，所以它可存取外圍物件的所有成員而無需添加任何飾詞。此外，inner classes 亦能

存取 enclosing class 的所有元素[3]。以下例子說明這一點：

```
//: c08:Sequence.java
// Holds a sequence of Objects.

interface Selector {
  boolean end();
  Object current();
  void next();
}

public class Sequence {
  private Object[] obs;
  private int next = 0;
  public Sequence(int size) {
    obs = new Object[size];
  }
  public void add(Object x) {
    if(next < obs.length) {
      obs[next] = x;
      next++;
    }
  }
  private class SSelector implements Selector {
    int i = 0;
    public boolean end() {
      return i == obs.length;
    }
    public Object current() {
      return obs[i];
    }
    public void next() {
      if(i < obs.length) i++;
    }
  }
```

[3] 這一點和 C++巢狀類別（*nested classes*）的設計極爲不同。在 C++中，那只是單純的名稱隱藏機制罷了，並不存在與外圍物件之間的連結關係，也沒有暗含的存取權。

```
  public Selector getSelector() {
    return new SSelector();
  }
  public static void main(String[] args) {
    Sequence s = new Sequence(10);
    for(int i = 0; i < 10; i++)
      s.add(Integer.toString(i));
    Selector sl = s.getSelector();
    while(!sl.end()) {
      System.out.println(sl.current());
      sl.next();
    }
  }
} ///:~
```

Sequence 只是一個大小固定的 **Object** array，以 class 形式加以包裝。你可以呼叫 **add()** 將新的 **Object** 加至該序列之末（前提是尚有空間）。如果想取得 **Sequence** 中的每個物件，有個名爲 **Selector** 的 interface，可讓你檢查目前是否已到序列末端（**end()**）、取得目前 **Object**（**current()**）、以及移至下一個 **Object**（**next()**）。由於 **Selector** 是個 interface，所以任何 classes 都可以以自己的方式來實作此一 interface，而許多函式都可以接受此一 interface 做爲引數，藉以產生一般化的程式碼。

在這裡，**SSelector** 是一個提供 **Selector** 機能的 **private** class。在 **main()** 中你可以看到，首先產生一個 **Sequence**，然後將一些 **String** 物件加入，接下來便經由呼叫 **getSelector()** 產生一個 **Selector**，再利用它來走訪 **Sequence** 並選擇其中的每一筆元素。

乍見之下，**SSelector** 只不過是個 inner class，但是讓我們更深入地觀察。注意，**end()**、**current()**、**next()** 等函式都用到了 **obs**，那是個 reference，並非 **SSelector** 的一部份，而是 enclosing class 的一個 **private** 資料成員。由於 inner class 可以存取 enclosing class 的所有成員，就像 inner class 自己擁有這些成員一樣，所以帶來很大的便利，一如上例所示。

換句話說 inner class 天生具有對 enclosing class 之所有成員的存取權力。這是怎麼做到的呢？inner class 必須記錄一個 reference，指向 enclosing

class 的某個特定物件。當你取用 enclosing class 的成員時，剛才說的那個（隱藏的）reference 便被用來選擇成員。很幸運地，編譯器會自動爲你處理所有細節。但你現在終於可以了解，inner class 物件被產生時，一定得關聯至其 enclosing class 的某個物件。建構 inner class 物件的同時，得有其 enclosing class 之物件（的 reference）才行。如果編譯器無法取得這麼一個 reference，便會給出錯誤訊息。大多數情況下這個過程無需程式員插手便會自動完成。

static inner classes（靜態內隱類別）

如果你不需要 inner class 物件和 enclosing class 物件之間的連結關係，你可以將 inner class 宣告爲 **static**。如果你想了解這麼宣告的確切時機，記住，一般的 inner class（譯註：也就是 non-**static** inner class）會自動記錄一個 reference 指向 enclosing class 的某個物件，而後者也就是此 inner class 物件的製造者。但是一旦你將 inner class 宣告爲 **static**，上述說法便不成立。**static** inner class 意味著：

1. 產生其物件時，並不需要同時存在一個 enclosing class 物件。
2. 你無法在 **static** inner class 物件中存取 enclosing class 物件。

static 和 non-**static** inner class 從另一個角度來看也不一樣。Non-**static** inner class 內的所有資料和函式都只能夠位於 class 的外層（outer level。譯註：此語應只是一種形容，不具實際技術意義），所以它不能夠擁有任何 **static** data、**static** fields、**static** inner class。然而 **static** inner classes 可以擁有那些東西：

```
//: c08:Parcel10.java
// Static inner classes.

public class Parcel10 {
  private static class PContents
  implements Contents {   // 譯註：Contents 見 p368
    private int i = 11;
    public int value() { return i; }
  }
```

```
  protected static class PDestination
      implements Destination {    // 譯註：Destination 見 p368
    private String label;
    private PDestination(String whereTo) {
      label = whereTo;
    }
    public String readLabel() { return label; }
    // Static inner classes can contain
    // other static elements:
    public static void f() {}
    static int x = 10;
    static class AnotherLevel {
      public static void f() {}
      static int x = 10;
    }
  }
  public static Destination dest(String s) {
    return new PDestination(s);
  }
  public static Contents cont() {
    return new PContents();
  }
  public static void main(String[] args) {
    Contents c = cont();
    Destination d = dest("Tanzania");
  }
} ///:~
```

main() 函式內完全不需要 **Parcel10** 物件；它只要採用一般用來選擇 **static** 成員的語法，呼叫「傳回 reference（指向 **Contents**）和 reference（指向 **Destination**）」的函式即可。

一如你將於稍後所見，non-**static** inner class 之中對於 enclosing class 物件的連結關係，是透過一個特殊的 **this** reference 形成。**static** inner class 不具此一特殊的 **this** reference，因而和 **static** 函式有些相似。

一般而言，你不能將任何程式碼置於 **interface** 內，但 **static** inner class 卻可以是 **interface** 的一部份。這是因爲 class 既然被宣告爲 **static**，也就不會破壞 interface 的規則 – **static** inner class 只不過是被置於 interface 的命名空間中罷了：

```java
//: c08:IInterface.java
// Static inner classes inside interfaces.

interface IInterface {
  static class Inner {
    int i, j, k;
    public Inner() {}
    void f() {}
  }
} ///:~
```

本書稍早我曾經建議你為每個 class 設計 **main()**，做為測試該 class 之用。這種作法的缺點之一就是，你得攜帶額外份量的編譯後程式碼。如果這對你來說是個問題，你可以運用 **static** inner class 來擺放你的測試碼：

```java
//: c08:TestBed.java
// Putting test code in a static inner class.

class TestBed {
  TestBed() {}
  void f() { System.out.println("f()"); }
  public static class Tester {
    public static void main(String[] args) {
      TestBed t = new TestBed();
      t.f();
    }
  }
} ///:~
```

這種作法會產生一個名為 **TestBed$Tester** 的獨立 class（如果想執行該程式，得輸入 **java TestBed$Tester** 才行）。你可以使用這個 class 來進行測試，但是在出貨產品中不需要含括此一 class。

取用（referring）outer class 的物件

在你需要產生一個 reference 指向 outer class 物件時，命名方式便是在 outer class 名稱之後緊接一個句點，然後再接 **this**。舉個例子，class **Sequence.SSelector** 內的任何函式都可以透過 **Sequence.this** 來產生一個 reference 指向 **Sequence**。產生出來的 reference 會自動被設定正確

型別。是的，編譯期即可得知確切型別並加以檢查，所以不會有執行期的額外負擔。

有時候，你會希望某個物件產生其本身內的某個 inner classes 物件。要達到這個目的，你得在 **new** 運算式中提供一份 reference，它得指向某個 outer class 物件，像這樣：

```
//: c08:Parcel11.java
// Creating instances of inner classes.

public class Parcel11 {
  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) {
    Parcel11 p = new Parcel11();
    // Must use instance of outer class
    // to create an instances of the inner class:
    Parcel11.Contents c = p.new Contents();
    Parcel11.Destination d =
      p.new Destination("Tanzania");
  }
} ///:~
```

如果你想直接產生 inner class 物件，你不能像你所（可能）想像地在 **new** 算式中使用 outer class 的名稱 **Parcel11**，你必須使用 outer class 物件來產生 inner class 物件，一如上例中的此行：

```
Parcel11.Contents c = p.new Contents();
```

因此，除非你已經擁有一個 outer class 物件，否則便無法產生其 inner class 物件。這是因爲 inner class 物件會被暗中連接到某個 outer class 物件

上，後者即該 inner class 物件的製造者。不過，如果你製作的是 **static** inner class，那就不需要一個 reference 指向 outer class 物件。

從多層巢狀 class 向外伸展觸角

[4]無論 inner class 被巢狀置放的層次有多深，其所有 outer classes 的成員都可被它存取，一如下例所示：

```
//: c08:MultiNestingAccess.java
// Nested classes can access all members of all
// levels of the classes they are nested within.

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();
  }
} ///:~
```

[4] 再次謝謝 Martin Danner。

你看到了，**MNA.A.B** 內可以直接呼叫 **g()** 和 **f()** 這兩個函式，無需添加任何飾詞（儘管它們實際上是 **private**）。這個例子同時也爲你示範如何在另一個 class 中產生「多層巢狀 inner classes 物件」。**.new** 語法會自動產生正確範疇（scope），所以呼叫建構式時你無需爲 class 名稱加上任何飾詞。

繼承 inner classes

由於 inner class 的建構式必須連接到一個 reference 指向 outer class 物件身上，所以當你繼承 inner class 時，事情便稍微複雜些。問題出在「指向 outer class 物件」的那個神秘 reference 必須被初始化，但 derived class 之內不存有可連結的預設物件。這個問題的答案是，使用專用語法，明確產生該關聯性：

```
//: c08:InheritInner.java
// Inheriting an inner class.

class WithInner {
  class Inner {}
}

public class InheritInner
    extends WithInner.Inner {
  //! InheritInner() {} // Won't compile
  InheritInner(WithInner wi) {
    wi.super();
  }
  public static void main(String[] args) {
    WithInner wi = new WithInner();
    InheritInner ii = new InheritInner(wi);
  }
} ///:~
```

你看到了，**InheritInner** 繼承的是 inner class 而非 outer class。但是當編譯至建構式時，*default* 建構式有問題；而且你也不能夠只是傳入一個 reference 指向 outer object，你還必須在建構式中使用以下語法：

```
enclosingClassReference.super();
```

這麼一來便能提供所需的 reference，而程式也能順利編譯下去。

inner classes 可被覆寫（overridden）嗎？

當你撰寫某個 inner class，然後再寫一個 class 繼承 enclosing class 並在自身之內重新定義那個 inner class，會發生什麼事？也就是說，我們能否覆寫（overriding）inner class？這似乎是個極具威力的概念，但是覆寫 inner class（好似把 inner class 視為 outer class 的一個函式），其實沒有什麼用處：

```
//: c08:BigEgg.java
// An inner class cannot be overriden
// like a method.

class Egg {
  protected class Yolk {
    public Yolk() {
      System.out.println("Egg.Yolk()");
    }
  }
  private Yolk y;
  public Egg() {
    System.out.println("New Egg()");
    y = new Yolk();
  }
}

public class BigEgg extends Egg { // 譯註：繼承 outer.
  public class Yolk {    // 譯註：這一段視為「覆寫 inner」.
    public Yolk() {
      System.out.println("BigEgg.Yolk()");
    }
  }
  public static void main(String[] args) {
    new BigEgg();
  }
} ///:~
```

注意，這個 *default* 建構式由編譯器自動合成，而且它必定會呼叫 base class 的 *default* 建構式。你可能會以為，由於 **main()** 所產生的乃是

BigEgg 物件，所以會使用覆寫過的 **Yolk**。事實並非如此，輸出如下：

```
New Egg()
Egg.Yolk()
```

這個範例告訴我們，當你繼承一個 outer class，不會有什麼神奇作用發生在 inner class 身上。上述兩個 inner classes 是完全獨立的個體，各有其專屬的命名空間。不過你還是可以明確指出想要繼承自 inner class：

```
//: c08:BigEgg2.java
// Proper inheritance of an inner class.

class Egg2 {
  protected class Yolk {
    public Yolk() {
      System.out.println("Egg2.Yolk()");
    }
    public void f() {
      System.out.println("Egg2.Yolk.f()");
    }
  }
  private Yolk y = new Yolk();
  public Egg2() {
    System.out.println("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() {
      System.out.println("BigEgg2.Yolk()");
    }
    public void f() {
      System.out.println("BigEgg2.Yolk.f()");
    }
  }
  public BigEgg2() { insertYolk(new Yolk()); }
  public static void main(String[] args) {
```

```
    Egg2 e2 = new BigEgg2();
    e2.g();
  }
} ///:~
```

現在，**BigEgg2.Yolk** 會明確繼承自 **Egg2.Yolk**，並覆寫其函式。**insertYolk()** 允許 **BigEgg2** 將它所擁有的 **Yolk** 物件向上轉型爲 **Egg2** 中的 **y** reference，所以當 **g()** 呼叫 **y.f()** 時，便會使用被覆寫過的 **f()**。輸出結果爲：

```
Egg2.Yolk()
New Egg2()
Egg2.Yolk()
BigEgg2.Yolk()
BigEgg2.Yolk.f()
```

注意，對 **Egg2.Yolk()** 的第二次呼叫，發生在 **BigEgg2.Yolk** 建構式中對其 base class 建構式的呼叫動作上。你看到了，當 **g()** 被呼叫，便會使用 **f()** 的覆寫版本。

Inner class 的識別符號（identifiers）

每個 class 都會產生一個 .class 檔，用來存放「生成此類物件時的所有必要資訊」。這些資訊會產生一個所謂的 **Class** 物件，這是一種 meta-class（譯註：請參考 p662）。你可能會猜想 inner classes 也必須產生 **.class** 檔來儲存其 **Class** 物件資訊。的確如此。這些檔案名稱與 classes 名稱之間有嚴謹的規則：先是 outer class 名稱，其後緊接 **'$'** 符號，然後再緊接 inner class 名稱。舉個例子，由 **InheritInner.java** 所產生的 **.class** 檔就有以下三個：

```
InheritInner.class
WithInner$Inner.class
WithInner.class
```

如果 inner class 沒有名稱，編譯器就自動產生數字，做爲 inner class 的識別符號。如果 inner classes 被巢狀置於其他 inner classes 內，其名稱就會直接附加於 **'$'** 符號與 outer class 識別符號（可能有多個）之後。

此種內部名稱產生方式不但單純而直覺，也十分穩當，而且能夠因應絕大多數情況[5]。由於這是 Java 的標準命名架構，所以產生出來的檔案自然而然與平台無關。（注意，爲了使這些檔案能夠運作，Java 編譯器會盡其所能地改變你的 inner classes）

爲什麼需要 inner classes？

現在，你已經看到了許多描述 inner classes 運作方式的語法與語意，但是這並無法回答 inner classes 的存在原因。爲什麼 Sun 要大費周章地加入這個語言功能呢？

一般來說，inner class 會繼承某個 class 或實作某個 interface，而且 inner class 內的程式碼會操作其 outer class 物件。所以你可以說，inner class 所提供的其實是針對 outer class 的某種窗口。

有個問題直指 inner class 的核心：如果我只需要「指向某個 **interface**」的 reference，爲什麼我不直接讓 outer class 實作該 **interface** 就好呢？答案是，如果這麼做就能符合你的需求，你的確應該這麼做。那麼，「由 inner class 實作 **interface**」和「由 outer class 實作 **interface**」兩者之間的區別究竟在那兒？答案是後者將使你無法總是享受到 **interfaces** 的便利性 — 有時候你得下探實作細目。所以，關於 inner classes 的存在，最讓人信服的理由是：

> 每個 inner class 都能夠各自繼承某一實作類別（implementation）。因此，inner class 不受限於 outer class 是否已繼承自某一實作類別。

如果少了 inner class 所提供的「繼承自多個具象（concrete）或抽象（**abstract**）類別」的能力，設計上和編程上的某些問題會變得十分棘手。所以，從某個角度來看 inner class，你可以說它是多重繼承問題的完整解決方案。interface 能夠解決其中一部份問題，但 inner classes 才能有

---

[5] '**$**' 符號同時也是 Unix Shell 中的 meta 字元，所以在列示 **.class** 檔案時，有時候會發生一些問題。對 Sun 這麼一個以 Unix 爲根據的公司來說，這種問題實在有點奇怪。我的猜測是，他們並沒有考慮這個問題，他們認爲你應該只專注源碼檔案。

效而實際地允許「多重實作繼承（multiple implementation inheritance）」。也就是說，inner classes 實際上允許你繼承多個 non-**interface**。

爲了更仔細地思考這個問題，請設想你目前擁有兩個 interfaces，它們必須以某種方式被實作於某個 class 內。由於 interfaces 所具備的彈性，你有兩個選擇：採用 single class 或 inner class：

```
//: c08:MultiInterfaces.java
// Two ways that a class can
// implement multiple interfaces.

interface A {}
interface B {}

class X implements A, B {}     // 譯註：狀況 1

class Y implements A {         // 譯註：狀況 2
  B makeB() {
    // Anonymous inner class:
    return new B() {};    // 譯註：{} 造成 inner class.
  }
}

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());
  }
} ///:~
```

當然，首先我們得假設，你的程式結構在上述任何一種方式下，都有其邏輯意義。通常你會從問題的本質得到決策上的某種導引，並以此選擇究竟

該使用 single class 或 inner class。如果沒有任何其他限制，從實作觀點來看，上例任何一種方式都沒有多大分別。兩種方式都行得通。

然而如果你所擁有的不是 **interfaces**，而是抽象或具象的 classes，你就必須使用 inner classes 來解決「多重繼承」的問題：

```
//: c08:MultiImplementation.java
// With concrete or abstract classes, inner
// classes are the only way to produce the effect
// of "multiple implementation inheritance."

class C {}
abstract class D {}

class Z extends C {
  // 譯註：以下的 {} 造成 inner class.
  D makeD() { return new D() {}; }
}

public class MultiImplementation {
  static void takesC(C c) {}
  static void takesD(D d) {}
  public static void main(String[] args) {
    Z z = new Z();
    takesC(z);
    takesD(z.makeD());
  }
} ///:~
```

如果你不需要解決「多重實作繼承」的問題，那麼你可以使用其他各種方法來撰寫程式，不需動用到 inner classes。但是，透過 inner classes，你可以擁有下列幾個額外性質：

1. inner class 可以擁有多份實體（instances），每個實體都擁有專屬的狀態資訊（state information），而這些資訊和 outer class 物件的資訊是相互獨立的。

2. 在單一 outer class 內你可以擁有多個 inner classes，每個都實作相同的 **interface**，或以不同方式繼承同一個 class。對此，稍後我會有一個範例說明。
3. 產生 inner class 物件的時間點，不見得必須和產生 outer class 物件同時。
4. outer class 和 inner class 之間不存在 ***is-a*** 的關係，inner class 是獨立個體。

舉個例子，如果 **Sequence.java** 不使用 inner class，那麼你就得宣稱「**Sequence** 是個 **Selector**」，而且對特定某個 **Sequence** 而言你只能擁有單一的 **Selector**。另外，如果你希望擁有第二個函式：**getRSelector()**，令它產生一個「回頭走」的 **Selector**，那麼你必須採用 inner class，才能有如此彈性。

Closures（終結）和Callbacks（回呼）

所謂 *closure* 是一種可被呼叫的物件，它會記錄一些資訊，這些資訊來自它的產生地所在的程式範疇（scope）。從這個定義來看，你可以發現，inner class 是一種 OO *closure*，因爲它不但包含來自 outer class 物件的種種資訊，而且還自動記錄一個「指向 outer class 物件」的 reference，並擁有操縱所有 outer class 成員的權限 — 即使面對 **private** 成員也不例外。

如果說有必要將指標機制含括到 Java 裡頭，最讓人信服的一個理由便是爲了提供所謂 *callbacks*（回呼）。在 callback 機制底下，某個物件被賦予一些資訊，這些資訊允許該物件在稍後某個時間點上呼叫原先的物件。一如你將在 13 章和 16 章所見，這的確是一種極爲有用的觀念。不過如果 callback 是以指標方式來完成，你就得指望程式員自己妥善處理，並且不至於誤用指標。截至目前我們所看到的是，Java 在設計上傾向更小心謹慎，所以指標並沒有被含括到這個程式語言內。

讓 inner class 提供 *closure* 功能，是完美的解決方案。比起指標來說，不僅更具彈性，而且安全許多。以下是一個簡單範例：

```
//: c08:Callbacks.java
// Using inner classes for callbacks
```

```
interface Incrementable {
  void increment();
}

// Very simple to just implement the interface:
class Callee1 implements Incrementable {
  private int i = 0;
  public void increment()
    i++;
    System.out.println(i);
  }
}

class MyIncrement {
  public void increment() {
    System.out.println("Other operation");
  }
  public static void f(MyIncrement mi) {
    mi.increment();
  }
}

// If your class must implement increment() in
// some other way, you must use an inner class:
class Callee2 extends MyIncrement {
  private int i = 0;
  private void incr()
    i++;
    System.out.println(i);
  }
  private class Closure implements Incrementable {
    public void increment() { incr(); }
  }
  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();
  }
} ///:~
```

這個例子進一步突顯了「在 outer class 中實作 interface」和「在 inner class 中實作 interface」之間的差異。就程式碼而言，**Callee1** 無疑是較簡單的方法。**Callee2** 繼承自 **MyIncrement**，後者擁有另一個不同的 **increment()**，這個函式會執行一些動作，而這些動作和 **Incrementable** interface 預期應該要做的事毫無關聯。當 **MyIncrement** 被 **Callee2** 繼承，你無法覆寫 **increment()** 以為 **Incrementable** 所用。所以你得利用 inner class 另行提供一份獨立的實作碼。請注意，當你撰寫 inner class 時，你並不會將任何東西加入 outer class 的介面，或修改該介面。

請注意，**Callee2** 內除了 **getCallbackReference()** 之外，都是 **private** 成員。如果想要建立與外界的連繫關係，interface **Incrementable** 是關鍵所在。在這個例子中，你可以看到 interfaces 如何允許介面與實作完全分離。

inner class **Closure** 很單純地藉由「實作 **Incrementable**」來提供與 **Callee2** 之間的關聯。這個關聯的確很安全。當然，不論誰取得了 **Incrementable** reference，它都只能呼叫 **increment()**，而不具備其他能力（這和指標不同；指標允許你進行任何動作）。

Caller 於其建構式中接受 **Incrementable** reference 做爲引數（儘管捕獲 callback reference 的動作隨時可能會發生），而且在某段時間之後，它會使用這個 reference 來「回頭呼叫」**Callee** class。

callback 的價值在於其彈性 — 你可以在執行時期動態決定究竟要呼叫哪個函式。到了 13 章，這個好處會更爲明顯。那一章處處使用 callbacks 來實現圖形使用介面（graphics user interface，GUI）的各種功能。

Inner classes ƒ¤ control frameworks

從以下即將介紹的所謂 *control framework*（控制框架）中，你可以獲得更爲具體的 inner classes 運用實例。

所謂 *application framework*（應用程式框架），是一組「被設計用來解決特定某種問題」的 classes。如果想套用某個 application framework，你得繼承一個或多個 classes，並覆寫其中某些函式。透過被覆寫的函式內的新版程式碼，便可將 application framework 所提供的通用解法特殊化，針對性地解決你的特定問題。所謂 control framework，其實就是一種特殊型式的 application framework，用來解決「事件（events）回應」的需要。一個系統如果主要工作在於回應諸般事件，我們稱爲「事件驅動系統（*event-driven system*）」。應用程式撰寫過程中最重要的一個題目便是圖形使用介面（Graphic User Interface，GUI），那幾乎完全是「事件驅動」方式。一如你將於第 13 章看到，Java Swing 程式庫便是一個 control framework，優雅解決了 GUI 問題，並大量採用 inner classes。

爲了觀察 inner class 究竟如何讓我們輕鬆建立並運用 control framework，讓我們假設，有個 control framework，其主要工作是在事件（events）狀態變成「就緒（ready）」時執行該事件。雖然「就緒」一詞有各種意義，但本例以 clock time 爲根據。接下來要說明的是，control framework 並未內含它所控制之事物的任何特定資訊。是的，會有一個介面用來描述每個

控制事件（control event），它將是個 **abstract** class 而非一個 **interface**，因為其預設行逕是「根據時間來執行控制動作」，所以可具有某種程度的實作：

```
//: c08:controller:Event.java
// The common methods for any control event.
package c08.controller;

abstract public class Event {
  private long evtTime;
  public Event(long eventTime) {
    evtTime = eventTime;
  }
  public boolean ready() {
    return System.currentTimeMillis() >= evtTime;
  }
  abstract public void action();
  abstract public String description();
} ///:~
```

上例建構式只是單純取得「你希望 **Event** 被執行起來」的時刻，**ready()** 會告訴你應該執行的時刻。當然，你也可以在 derived class 中覆寫 **ready()**，不再以時刻為依據來決定事件是否該執行。

當 **Event** 就緒（**ready()**）時便會呼叫 **action()**。透過 **description()** 則可取得 **Event** 相關文字訊息。

以下檔案內含一個實際的 control framework，可以管理並觸發事件。第一個 class 實際上只是個輔助用的 class（一個 helper），其職責是儲存 **Event** 物件。你可以使用任何適當的容器加以替換。你會發現，第 9 章的各種容器都是適當的候選人，你不需要自己寫一個。

```
//: c08:controller:Controller.java
// Along with Event, the generic
// framework for all control systems:
package c08.controller;

// This is just a way to hold Event objects.
class EventSet {
  private Event[] events = new Event[100];
```

```
  private int index = 0;
  private int next = 0;
  public void add(Event e) {
    if(index >= events.length)
      return; // (In real life, throw exception)
    events[index++] = e;
  }
  public Event getNext() {
    boolean looped = false;
    int start = next;
    do {
      next = (next + 1) % events.length;
      // See if it has looped to the beginning:
      if(start == next) looped = true;
      // If it loops past start, the list
      // is empty:
      if((next == (start + 1) % events.length)
         && looped)
        return null;
    } while(events[next] == null);
    return events[next];
  }
  public void removeCurrent() {
    events[next] = null;
  }
}

public class Controller {
  private EventSet es = new EventSet();
  public void addEvent(Event c) { es.add(c); }
  public void run() {
    Event e;
    while((e = es.getNext()) != null) {
      if(e.ready()) {
        e.action();
        System.out.println(e.description());
        es.removeCurrent();
      }
    }
  }
} ///:~
```

EventSet 能儲存 100 個 **Events**。如果此處使用第 9 章所說的容器，你就不需要再擔心其最大容量，因爲那些容器能夠自己調整大小。**index** 被用來記錄下一個可用空間。當你尋找 list 中的下一個 **Event** 時，得使用 **next** 來檢查是否已經繞行一周。這在呼叫 **getNext()** 時尤其重要，因爲當我們執行完 **Event** 物件之後便得運用 **removeCurrent()** 將這些 **Event** 物件移除，所以 **getNext()** 會在 list 的走訪過程中遇上一些「空洞」。

請注意，**removeCurrent()** 並不會設定某種旗標以表示某個物件不再被使用。相反地，它會將 reference 設爲 **null**。這麼做很重要，因爲如果垃圾回收器看到某個仍在使用中的 reference，它就不會清理該物件。如果你認爲你的 reference 可能散於四處，那麼將它們設爲 **null** 是個不錯的方式，讓垃圾回收器有權將它們清理掉。

Controller 是執行實際工作的地方。它使用 **EventSet** 來儲存 **Event** 物件，並提供 **addEvent()** 讓你將新事件（events）加入 list。最重要的是 **run()**，它會走訪 **EventSet**，尋找已經就緒而可被執行的 **Event** 物件，然後呼叫其 **action()**，並利用 **description()** 列印說明訊息，再將該 **Event** 自 list 中移除。

請注意，上述這份設計，到目前爲止，對於 **Event** 實際上會做什麼事，你其實是一無所知的。這也正是這份設計的關鍵所在；你看它是如何地隔離了「可能更動的事物」和「不變的事物」。或者，以我的詞彙來說是，變化發生於「各種 **Event** 物件之間彼此互異的行爲模式」上，而你解釋這些不同行爲模式的方式就是：撰寫不同的 **Event** subclasses。

這正是 inner classes 派上用場的地方。它們允許兩件事情：

1. 在單一 class 內產生 control framework 的完整實作內容，並因而將專屬於該實作方式的所有資訊都封裝起來。inner classes 則用以表現「解決問題所必要的」多個不同種類的 **action()**。此外，稍後例子還運用 **private** inner class，使整個實作內容被完全隱藏起來，因而即使有所改變也不會帶來傷害。

2. inner classes 能夠使實作手法不至於太拙劣，因爲你可以輕易存取 outer class 的所有成員。一旦缺乏這種能力，程式碼可能會一團糟，導致你必須尋找其他替代方案。

請試著設想某個特殊 control framework 的實作內容 — 它被設計用來控制溫室的機能[6]。每個動作彼此之間完全不同：燈光控制、水、調溫器開關、鳴鈴、系統重啓。但是 control framework 被設計用來將這些不同的程式碼輕易隔離開來。inner classes 讓你得以在單一 class 內爲同一個 base class（**Event**）提供多份衍生版本。面對每一種類型的行爲，你都繼承一個新的 **Event** inner class，並在其 **action()** 內撰寫控制碼。

做爲一個典型的的 application framework，**GreenhouseControls** 繼承自 **Controller**：

```
//: c08:GreenhouseControls.java
// This produces a specific application of the
// control system, all in a single class. Inner
// classes allow you to encapsulate different
// functionality for each type of event.
import c08.controller.*;

public class GreenhouseControls
    extends Controller {
  private boolean light = false;
  private boolean water = false;
  private String thermostat = "Day";
  private class LightOn extends Event {
    public LightOn(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here to
      // physically turn on the light.
      light = true;
```

[6]基於某種理由，這一直都是我很樂意解決的一個問題。它出自於我先前的一本書《*C++ Inside & Out*》，不過 Java 提供了更優雅的解決方案。

```
    }
    public String description() {
      return "Light is on";
    }
  }
  private class LightOff extends Event {
    public LightOff(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here to
      // physically turn off the light.
      light = false;
    }
    public String description() {
      return "Light is off";
    }
  }
  private class WaterOn extends Event {
    public WaterOn(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here
      water = true;
    }
    public String description() {
      return "Greenhouse water is on";
    }
  }
  private class WaterOff extends Event {
    public WaterOff(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here
      water = false;
    }
    public String description() {
      return "Greenhouse water is off";
    }
```

```
  }
  private class ThermostatNight extends Event {
    public ThermostatNight(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here
      thermostat = "Night";
    }
    public String description() {
      return "Thermostat on night setting";
    }
  }
  private class ThermostatDay extends Event {
    public ThermostatDay(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here
      thermostat = "Day";
    }
    public String description() {
      return "Thermostat on day setting";
    }
  }
  // An example of an action() that inserts a
  // new one of itself into the event list:
  private int rings;
  private class Bell extends Event {
    public Bell(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Ring every 2 seconds, 'rings' times:
      System.out.println("Bing!");
      if(--rings > 0)
        addEvent(new Bell(
          System.currentTimeMillis() + 2000));
    }
    public String description() {
      return "Ring bell";
```

```
    }
  }
  private class Restart extends Event {
    public Restart(long eventTime) {
      super(eventTime);
    }
    public void action() {
      long tm = System.currentTimeMillis();
      // Instead of hard-wiring, you could parse
      // configuration information from a text
      // file here:
      rings = 5;
      addEvent(new ThermostatNight(tm));
      addEvent(new LightOn(tm + 1000));
      addEvent(new LightOff(tm + 2000));
      addEvent(new WaterOn(tm + 3000));
      addEvent(new WaterOff(tm + 8000));
      addEvent(new Bell(tm + 9000));
      addEvent(new ThermostatDay(tm + 10000));
      // Can even add a Restart object!
      addEvent(new Restart(tm + 20000));
    }
    public String description() {
      return "Restarting system";
    }
  }
  public static void main(String[] args) {
    GreenhouseControls gc =
      new GreenhouseControls();
    long tm = System.currentTimeMillis();
    gc.addEvent(gc.new Restart(tm));
    gc.run();
  }
} ///:~
```

請注意，**light**、**water**、**thermostat** 和 **rings** 通通屬於 outer class **GreenhouseControls**；inner class 可以存取這些資料成員而無需添加任何飾詞，也無需任何特別的存取權限。此外大多數 **action()** 都和某種硬體控制有關，而這些硬體控制極有可能得呼叫 non-Java 程式碼。

大多數 **Event** classes 看起來十分類似，但 **Bell** 和 **Restart** 較為特別。**Bell** 會發出響聲，而且如果未能持續鳴響足夠的時間，它便會將新的 **Bell** 物件加至事件串列（event list）中，所以稍後便會再度鳴響。請注意 inner classes 看起來多麼像多重繼承：**Bell** 具有 **Event** 的所有函式，而它似乎也擁有 outer class **GreenhouseControls** 的所有函式。

Restart 負責將系統初始化，所以它會將所有合適的事件通通加入。當然，欲達成這個目的，有個更具彈性的方法，那便是避免將事件寫死在程式碼中，而是改由檔案讀出（第 11 章有個練習，就是希望你修改這個範例以達上述目標）。由於 **Restart()** 只是另一個 **Event** 物件，所以你也可以在 **Restart.action()** 中將 **Restart** 物件加入，藉以使得系統能夠定期重新啟動自己。你只需在 **main()** 中產生 **GreenhouseControls** 物件，然後加入 **Restart** 物件使其開始運作即可。

這個例子應該可以讓你深刻體會 inner class 的價值，特別是將它應用於 control framework 時。第 13 章還會讓你知道，inner classes 被用來描述圖形使用介面（GUI）的行為模式時，有多麼優雅。一旦你讀完該章，應該就會完全信服 inner class 的價值了。

摘要

相較於許多 OOP 語言所具備的其他觀念，interfaces 和 inner classes 更為複雜。C++之中並沒有兩者的類似機制。將兩者合併起來可以解決 C++ 透過多重繼承（multiple inheritance，MI）所嘗試解決的問題。C++ 多重繼承用起來十分困難，對照起來，Java 的 interfaces 和 inner classes 比較容易些。

雖然這兩個性質相當直觀而簡單，但它們的運用屬於設計上的議題，這一點和多型（polymorphism）極為相像。使用一段時間之後，你就愈來愈能判斷，什麼場合應該使用 interface、什麼場合應該使用 inner classes、什麼場合應該二者併用。此刻，你應該至少能夠輕鬆掌握其語法和語意。一旦你看到了這兩個性質被實際用上，最終便能融會貫通。

練習

某些經過挑選的題目，其解答置於《*The Thinking in Java Annotated Solution Guide*》電子文件中。僅需小額費用便可自 *www.BruceEckel.com*.網站取得。

1. 請證明 **interface** 內的資料成員會被自動設爲 **static** 和 **final**。

2. 撰寫一個 **interface**，內含三個函式，然後在另一個不同的 **package** 中實作此一 **interface**。

3. 證明 **interface** 內的所有函式都會被自動設爲 **public**。

4. 在 **c07:Sandwich.java** 內撰寫名爲 **FastFood** 的 interface（並賦予它適宜的函式），然後修改 **Sandwich**，使它實作出 **FastFood**。

5. 撰寫三個 **interfaces**，每一個都擁有兩個函式。撰寫一個新的 **interface**，繼承上述三者，並新增一個函式。撰寫一個 class 實作出那個新的 interface，並繼承另一個 concrete class（具象類別）。接下來寫出四個函式，各自接受上述四個 **interfaces** 之一做爲引數。在 **main()** 中，爲你的那個 class 產生一個物件，並將它傳入上述四個函式。

6. 修改練習 5：撰寫一個 **abstract** class（抽象類別），並繼承此一 class 以獲得 derived class。

7. 修改 **Music5.java**：加入一個 **Playable interface**。將 **play()** 的宣告自 **Instrument** 移除。將 **Playable** 放進 **implements** list 中，藉以將 **Playable** 加至 derived class 內。修改 **tune()** 使它接受 **Playable** 做爲引數（取代原先的 **Instrument**）。

8. 修改第 7 章的練習 6，使 **Rodent** 成爲一個 **interface**。

9. 在 **Adventure.java** 內加入名爲 **CanClimb** 的 **interface**。讓這個 interface 依循其他 interfaces 的形式。

10. 撰寫程式，在其中匯入（imports）**Month2.java**，並加運用。

11. 依循 **Month2.java** 所給的範例，撰寫每週各日的列舉（enum）集合。

12. 撰寫一個 **interface** 並令它擁有一個以上的函式。在另一個 package 內撰寫另一個 class，並在其中加入一個 **protected** inner class，實作出上述 **interface**。在第三個 package 內，繼承你自己撰寫的那個 class，並在某個函式中回傳上述那個 **protected** inner class 的物件。請在回傳時將此物件向上轉型至其 base **interface**。

13. 撰寫一個 **interface** 並令它具備一個以上的函式，並在某個函式中定義 inner class 以實作出該 **interface**。inner class 必須傳回一個 reference，指向那個 **interface**。

14. 重覆練習 13，改在函式內的某個程式範疇（scope）中定義 inner class。

15. 重覆練習 13，改用匿名的 inner class。

16. 撰寫一個「實作出 **public** interface」的 **private** inner class。為它撰寫函式，使後者傳回一個 reference，指向該 **private** inner class 的一個實體（instance），並將它向上轉型至該 **interface**。試著向下轉型至 inner class，藉此證明 inner class 被完全隱藏起來。

17. 撰寫一個 class，使之擁有 non-*default* 建構式，並且不具任何 *default* 建構式。撰寫第二個 class，使其函式回傳一個 reference 指向第一個 class。撰寫一個匿名的 inner class 繼承自第一個 class，並為它產生一個物件做為函式回傳值。

18. 撰寫一個 class 具備 **private** 資料成員和 **private** 函式。撰寫一個 inner class，使其函式修改 outer class 資料成員，並呼叫 outer class 函式。請在第二個 outer class 函式中產生 inner class 物件，並呼叫其函式。試說明 outer class 物件身上發生的效應。

19. 重覆練習 18，改而使用匿名的 inner class。

20. 撰寫含有 static inner class 的 class。在 **main()** 中產生其實體。

21. 撰寫含有 **static** inner class 的一個 **interface**。實作它，並產生該 inner class 的實體（instance）。

22. 撰寫一個 class，內含一個 inner class。後者本身尚且含有另一個 inner class。重覆上述要求，但改用 **static** inner class。請注意，**.class** 檔的檔名由編譯器產生。

23. 撰寫一個 class，內含一個 inner class。在另一個 class 中產生此 inner class 的實體（instance）。

24. 撰寫一個 class，內含一個 inner class，後者擁有 non-*default* 建構式。撰寫第二個具備 inner class 的 class，令其 inner class 繼承第一個 inner class。

25. 修正 **WindError.java** 中的問題。

26. 修改 **Sequence.java**，加入 **getRSelector()**，後者回傳 **Selector interface** 的另一份實作品：能夠從末端移至前端，以反向方式走訪此一序列。

27. 撰寫 **interface U** 並使之具備三個函式。撰寫 class **A** 並使之具備一個函式，此函式透過匿名的 inner class，產生一個 reference 指向 **U**。撰寫 class **B**，令它內含一個 **U** array。**B** 擁有一個函式，接受一個 reference 指向「**U** array 中的某個 **U**」，並將之儲存起來。**B** 的第二個函式則是將 array 中的某個 reference（由函式引數指定）設為 **null**。第三個函式走訪 array，並呼叫 **U** 中的函式。請在 **main()** 中產生一組 **A** 物件和一個 **B** 物件。將 **A** 物件所形成的 **U** references 填入 **B** 中。使用 **B** 回呼（*call back*）所有 **A** 物件。並嘗試移去 **B** 中的某些 references。

28. 在 **GreehhouseControls.java** 內加入一個「風扇開啓、關閉」的 **Event** inner class。

29. 證明「inner class 具備對其 outer class 之 **private** 元素的存取權限」。判斷反向是否成立。

9: 持有你的物件

這是個十分簡單的計劃：一群固定數量且壽命已知的物件（objects）。

一般而言，你的程式總是會根據某些條件來產生新的物件，而這些條件只有在程式執行時才有辦法知道。不到執行期，無法得知究竟需要多少數量的物件，也無法知道這些物件的確切型別。爲了解決這個常見的編程問題，你必須有能力在任意時刻、任何地點產生任意個數的物件。因此你無法只仰賴具名的 reference 來持有物件，像這樣：

```
MyObject myReference;
```

因爲你永遠不知道，實際上需要多少個這樣的物件。

爲了解決這個十分基本的問題，Java 提供了物件（或者應該說是物件的 reference）的數種持有方式。其中屬於語言內建者，是先前介紹過的 array。此外 Java 公用程式庫（utilities）也提供了一組相當完整的容器類別（*container classes*），又稱爲群集類別（*collection classes*），但因 Java 2 程式庫已經使用 **Collection** 一詞代表程式庫中的某個子集，所以我習慣使用的詞彙是「容器」。針對物件的持有和操作，容器提供了極爲精巧的作法。

Arrays（陣列）

第 4 章最末一節已經涵蓋了 array 的必要簡介。該節說明 array 的定義和初始化方式。至於物件的持有，則是本章焦點所在。array 不過是持有物件的方式之一罷了，另有其他多種物件持有方式。那麼，array 的特色在哪裡？

關於 array 和其他容器之間的區別，存在兩個議題：效率和型別。array 是 Java 用來「儲存及隨機存取一連串物件（其實是物件的 references）」的各種作法中，最有效率的一種。array 是個極簡單的線性序列，其中元素能夠被快速存取。不過效率帶來的犧牲是：當你產生 array 時，其容量固定且無法動態改變。你可能會想要產生某個固定容量的 array，在空間不敷使用時再產生另一個新 array，並將舊 array 中的 references 全部搬到新 array 中。此即 **ArrayList** class 的運作方式，本章稍後會介紹它。不過由於這種容量上的彈性必須付出額外代價，**ArrayList** 的效率明顯比 array 差。

C++ 的 **vector** 容器確切知道它所持有的物件隸屬什麼型別（譯註：目前的 JDK1.4 也辦得到），但它和 Java array 相較有個缺點：C++ **vector** 的 **operator[ ]** 並不進行邊界檢查，所以你的存取動作可以超越其尾端[1]。然而 Java 之中無論是 array 或其他容器都會進行邊界檢查，一旦越界就會出現 **RuntimeException**。一如你將於第 10 章看到，這類異常表示錯誤由程式員造成，因此你不需要自行做邊界檢查。讓我說點題外話，C++ **vector** 之所以不在每次存取時進行邊界檢查，是基於效率考量；是的，Java array 及其他容器都會因爲邊界檢查而帶來額外的效率負擔。

本章還會討論其他泛型容器，包括 **List**、**Set**、**Map**。它們不會以任何特定型別來看待它們所持有的物件。或者說，它們將持有物件一律視爲 **Object** 型別 — 這是所有 Java classes 的根類別。從某個觀點來看，這種運作方式很好：只要產生單一容器，便可將任意 Java 物件置於其中（基本型別（primitive types）除外，但你可以將它們視爲常數，透過 Java 所提供的外覆類別（wrapper）將它們置於容器內，或透過你自己撰寫的 class 加以包裝，使其值可變動）。這正是 array 比泛型容器優越的第二點：當你產生 array 時，它持有的是特定型別的物件。這意味編譯期的型別檢查會防止你將不正確的型別置入 array，或擷取出不正確的型別。當然，Java 最終還是能夠（在編譯期或執行期）預防你將不正確的訊息發送給物件。

[1] 不過，你可以查詢 vector 的容量，而且 at() 會進行邊界檢查。

所以並沒有哪一種方法比較不安全。只不過如果編譯器能夠爲你指出問題所在，總是比較好，程式的執行速度也會快些。而且使用者比較沒有機會對異常（exceptions）發出驚訝聲。

基於效率和型別檢驗兩個理由，如果可以的話你應該儘可能使用 array。不過當你試著解決更一般化的問題時，array 的功能就可能過於受限。本章討論完 array 之後，剩餘部份將致力於介紹 Java 的容器類別。

Arrays 是第一級物件（first-class objects）

不論你所使用的 array 型別爲何，array 名稱本身實際上是個 reference，指向 heap 之內的某個實際物件。這個物件持有「指向其他物件」的一些 references。這個物件可經由「array 初始化語法」被自動產生，也可以以 **new** 運算式手動產生。array 物件之內有個名爲 **length** 的唯讀成員，它能夠告訴你 array 物件內的元素個數。除此之外，你唯有透過 '**[ ]**' 語法才能取用 array 物件。

> 譯註：請注意，本節所保留的英文術語中，*array object* 是指表現「array 本身」的那個物件，*objects array* 是指「由物件形成的 array」。*primitives array* 是指由隸屬基本型別之元素所構成的 array。

下面這個例子示範各種不同形式的 array 初始化動作，以及 array references 被指派至另一個不同的 *array objects* 的方式。這個例子同時也說明了，*objects array* 和 *primitives array* 在運用上幾乎一模一樣。唯一差別在於，前者持有的是 references，後者直接持有基本型別之值。

```
//: c09:ArraySize.java
// Initialization & re-assignment of arrays.

class Weeble {} // A small mythical creature

public class ArraySize {
  public static void main(String[] args) {
    // Arrays of objects:
    Weeble[] a; // Null reference
    Weeble[] b = new Weeble[5]; // Null references
```

```
    Weeble[] c = new Weeble[4];
    for(int i = 0; i < c.length; i++)
      c[i] = new Weeble();
    // Aggregate initialization:
    Weeble[] d = {
      new Weeble(), new Weeble(), new Weeble()
    };
    // Dynamic aggregate initialization:
    a = new Weeble[] {
      new Weeble(), new Weeble()
    };
    System.out.println("a.length=" + a.length);
    System.out.println("b.length = " + b.length);
    // The references inside the array are
    // automatically initialized to null:
    for(int i = 0; i < b.length; i++)
      System.out.println("b[" + i + "]=" + b[i]);
    System.out.println("c.length = " + c.length);
    System.out.println("d.length = " + d.length);
    a = d;
    System.out.println("a.length = " + a.length);

    // Arrays of primitives:
    int[] e; // Null reference
    int[] f = new int[5];
    int[] g = new int[4];
    for(int i = 0; i < g.length; i++)
      g[i] = i*i;
    int[] h = { 11, 47, 93 };
    // Compile error: variable e not initialized:
    //!System.out.println("e.length=" + e.length);
    System.out.println("f.length = " + f.length);
    // The primitives inside the array are
    // automatically initialized to zero:
    for(int i = 0; i < f.length; i++)
      System.out.println("f[" + i + "]=" + f[i]);
    System.out.println("g.length = " + g.length);
    System.out.println("h.length = " + h.length);
    e = h;
    System.out.println("e.length = " + e.length);
    e = new int[] { 1, 2 };
```

```
    System.out.println("e.length = " + e.length);
  }
} ///:~
```

下面是程式的輸出：

```
b.length = 5
b[0]=null
b[1]=null
b[2]=null
b[3]=null
b[4]=null
c.length = 4
d.length = 3
a.length = 3
a.length = 2
f.length = 5
f[0]=0
f[1]=0
f[2]=0
f[3]=0
f[4]=0
g.length = 4
h.length = 3
e.length = 3
e.length = 2
```

array **a** 會被初始化爲一個 **null** reference。除非你給它適當初值，否則編譯器不會允許你對它進行任何動作。array **b** 被初始化，指向一個以 **Weeble** references 構成的 array，但並沒有任何實際的 **Weeble** 物件被置於其中。不過你還是可以查詢 array 的容量，因爲 **b** 已經指向一個合法的 *array object*。這也指出了一個小小缺點：你無法判斷 array 之中實際存有多少元素，因爲 **length** 只能告訴你 array 的容量，而非實際元素個數。不過在 *array object* 被產生的同時，其中所有 references 都會被初始化爲 **null**，所以只要檢查某個位置是否爲 **null**，便可判斷該位置是否持有物件。類似情況，*primitives array* 會將數值元素自動初始化爲 0，將 **char** 元素自動初始化爲 **(char) 0**，並將 **boolean** 元素自動初始化爲 **false**。

array **c** 示範如何在產生 *array object* 之後緊接著將 **Weeble** 物件塞入所有位置。array **d** 示範「聚集初始化（aggregate initialization）」語法，此語法能夠產生 *array object*（自動地在 heap 上進行 **new**，就和 array **c** 一樣），並給定初始的 **Weeble** 物件，所有動作以一行述句完成。

下一個 array 初始化動作，可被視爲一種「動態的聚集初始化（dynamic aggregate initialization）」。前述 **d** 所使用的聚集初始化方式，必須在 **d** 的定義點進行。但透過第二種語法，便可在任意地點產生並初始化 *array object*。舉個例子，假設 **hide()** 接收 **Weeble** *objects array*，你可以這麼呼叫之：

```
hide(d);
```

也可以動態產生那個將被傳入做爲引數的 array：

```
hide(new Weeble[] { new Weeble(), new Weeble() });
```

某些情況下，這個新語法可以爲程式碼的撰寫提供更便捷的途徑。

底下這行算式：

```
a = d;
```

示範如何將「已經指向某個 *array object*」的 reference 指派至另一個 *array object*，動作和其他型別的 object reference 沒有任何不同。現在，**a** 和 **d** 指向 heap 上的同一個 *array object*。

ArraySize.java 的第二部份告訴我們，*primitive array* 的運作方式和 *object array* 類似，只不過前者直接持有基本型別的元素值罷了。

容納基本型別元素的容器（Containers of primitives）

容器類別（container classes）僅能持有 references（指向物件）。但面對 array，我們卻可以產生直接持有基本型別數值的 array（譯註：所謂 *primitives array*），也可以產生持有 references（全都指向物件）的 array（譯註：所謂 *objects array*）。我們可以使用外覆類別（wrapper，例如 **Integer**, **Double**）將基本型別值置於容器中。但是這些外覆類別使用起來可能不很容易上手。此外，*primitives array* 的效率比起「容納基本

型別之外覆類別（的 reference）」的容器好太多了（你可稱那種容器爲 *wrapped primitives containers*）。

當然，如果你的操作對象是基本型別，而且需要在空間不足時自動擴增容量，array 便不適合，此時就得使用外覆類別的容器了。你可能會認爲應該針對各種基本型別都提供一份特殊版的 **ArrayList**，但 Java 並未如此。有朝一日，某種模版機制（templatizing mechanism）也許能幫助 Java 更妥善地處理此一問題[2]。

回傳一個 array

假設你正在撰寫某個函式，你希望它不只是回傳單一數值，而是回傳一大串值。C/C++之類的程式語言會讓這個問題變得十分困難，因爲你無法單單只回傳 array，你得回傳一個指向 array 的指標。這種作法會帶來問題，因爲 array 壽命的控制變得極其麻煩，很容易導致記憶體漏洞。

Java 採取類似手法，但你只要回傳 array 就行了。當然，實際回傳的是個 reference，指向一個 array，但此刻你不需要負起照料 array 的責任 — 只要你還需要它，它就會持續存在。當你不再需要它，垃圾回收器會處理。

以下便是一例，回傳一個 **String** array：

```
//: c09:IceCream.java
// Returning arrays from methods.

public class IceCream {
  static String[] flav = {
    "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie"
  };
  static String[] flavorSet(int n) {
```

[2] 這是 C++明顯優於 Java 之處。C++ 經由關鍵字 **template** 支援所謂「參數化型別」（*parameterized types*）（譯註：此一想法已於 JDK1.4 實現，採用的技術不同於 C++）

```
    // Force it to be positive & within bounds:
    n = Math.abs(n) % (flav.length + 1);
    String[] results = new String[n];
    boolean[] picked =
      new boolean[flav.length];
    for (int i = 0; i < n; i++) {
      int t;
      do
        t = (int)(Math.random() * flav.length);
      while (picked[t]);
      results[i] = flav[t];
      picked[t] = true;
    }
    return results;
  }
  public static void main(String[] args) {
    for(int i = 0; i < 20; i++) {
      System.out.println(
        "flavorSet(" + i + ") = ");
      String[] fl = flavorSet(flav.length);
      for(int j = 0; j < fl.length; j++)
        System.out.println("\t" + fl[j]);
    }
  }
} ///:~
```

flavorSet() 會產生名爲 **results** 的一個 **String** array，其容量爲 **n**，由函式的引數決定。接下來它會隨機從 array **flav** 中選出各種味道（flavor），並將選出值置於 **results** 內，最終便是回傳 **results**。回傳 array 的動作就跟回傳其他物件一樣，畢竟 array 只是一個 reference。被回傳的 array 究竟是在 **flavorSet()** 內產生的或是在其他地方產生，一點都不重要。當你不再使用這個 array，垃圾回收器便會清理它。當你還需要它時，它便會持續存活。

題外話，請注意，當 **flavorSet()** 隨機選擇各種味道時，它會確保絕不做二次重複挑選。這種保證是因爲 **do** 迴圈不斷進行隨機選取，直至找到一個不在 **picked** array 中的元素爲止（當然你也可以採用「字串比較」的方式來檢查隨機選取的結果是否已在 **results** array 內。不過「字串比較」法

的效率並不好）。如果找到了符合條件者，便將結果加入，然後再繼續尋找下一個（**i** 值會遞增 1）。

main() 會印出 20 組味道。所以你可以觀察到，**flavorSet()** 每次皆以隨機方式挑選各種不同的味道。如果你將輸出結果導至檔案，便可輕易進行觀察。當你檢視檔案內容時，請千萬記住，你只是「想要」冰淇淋，但並不「需要」它。

Arrays class

你可以在 **java.util** 中找到 **Arrays** class，它擁有一組 **static** 函式，能夠執行許多 array 公用函式。計有四個基本函式：**equals()** 用來比較兩個 array 是否相等；**fill()** 用來將某值填入 array 內；**sort()** 進行 array 的排序；**binarySearch()** 在已排序的 array 中尋找元素。所有函式皆被重載，可用於所有基本型別和 **Objects** 身上。此外還有一個 **asList()** 函式，接受任意 array，並將它轉換爲 **List** 容器 — 此容器將在本章稍後出現。

雖然 **Array** 十分有用，但它缺乏完整功能。舉個例子，如果能夠輕易印出 array 的每個元素，不必每次撰寫 **for** 迴圈來處理，其實滿好的。而且如你所見，**fill()** 僅接收單一值，然後將之填入 array 內，那麼如果你想要以隨機亂數值填滿 array，**fill()** 便幫助不大。

因此爲 **Arrays** 補充一些函式是有實際意義的。我把它們置於 **pacckage com.bruceeckel.util** 中。它們可以印出任意型別的 array，也可以將所有 *generator*（產生器）所產生的數值（或物件）填入 array 內。此一產生器可由你自行定義。

由於程式碼得處理 **Object** 型別和各種基本型別，所以有大量幾近重複的程式碼出現[3]。例如我得爲每一種型別提供一份 "generator" interface，因爲在不同型別的情況下，**next()** 的回傳值型別都不相同。

---

[3] C++ 程式員會注意到，如果使用預設引數（default arguments）和模版（template），程式碼能夠收斂到怎樣的程度。Python 程式員則會注意到，這整個程式庫在該語言中大部份都是沒有必要的。

```
//: com:bruceeckel:util:Generator.java
package com.bruceeckel.util;
public interface Generator
  Object next();
} ///:~

//: com:bruceeckel:util:BooleanGenerator.java
package com.bruceeckel.util;
public interface BooleanGenerator {
  boolean next();
} ///:~

//: com:bruceeckel:util:ByteGenerator.java
package com.bruceeckel.util;
public interface ByteGenerator {
  byte next();
} ///:~

//: com:bruceeckel:util:CharGenerator.java
package com.bruceeckel.util;
public interface CharGenerator {
  char next();
} ///:~

//: com:bruceeckel:util:ShortGenerator.java
package com.bruceeckel.util;
public interface ShortGenerator {
  short next();
} ///:~

//: com:bruceeckel:util:IntGenerator.java
package com.bruceeckel.util;
public interface IntGenerator {
  int next();
} ///:~

//: com:bruceeckel:util:LongGenerator.java
package com.bruceeckel.util;
public interface LongGenerator {
  long next();
} ///:~
```

```
//: com:bruceeckel:util:FloatGenerator.java
package com.bruceeckel.util;
public interface FloatGenerator {
  float next();
} ///:~

//: com:bruceeckel:util:DoubleGenerator.java
package com.bruceeckel.util;
public interface DoubleGenerator {
  double next();
} ///:~
```

Array2 含有許多 **print()** 函式，分別針對各種型別進行重載。你可以單純印出 array、或是在 array 被印出前加一段訊息，或是印出 array 中某個範圍內的元素。至於 **print()** 函式碼本身無需多加解釋，你也能明白：

```
//: com:bruceeckel:util:Arrays2.java
// A supplement to java.util.Arrays, to provide
// additional useful functionality when working
// with arrays. Allows any array to be printed,
// and to be filled via a user-defined
// "generator" object.
package com.bruceeckel.util;
import java.util.*;

public class Arrays2 {
  private static void
  start(int from, int to, int length) {
    if(from != 0 || to != length)
      System.out.print("["+ from +":"+ to +"] ");
    System.out.print("(");
  }
  private static void end() {
    System.out.println(")");
  }
  public static void print(Object[] a) {
```

```
    print(a, 0, a.length);
  }
  public static void
  print(String msg, Object[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(Object[] a, int from, int to){
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to -1)
        System.out.print(", ");
    }
    end();
  }
  public static void print(boolean[] a) {
      print(a, 0, a.length);
  }
  public static void
  print(String msg, boolean[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(boolean[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to -1)
        System.out.print(", ");
    }
    end();
  }
  public static void print(byte[] a) {
      print(a, 0, a.length);
  }
  public static void
  print(String msg, byte[] a) {
    System.out.print(msg + " ");
```

```
    print(a, 0, a.length);
  }
  public static void
  print(byte[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to -1)
        System.out.print(", ");
    }
    end();
  }
  public static void print(char[] a) {
      print(a, 0, a.length);
  }
  public static void
  print(String msg, char[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(char[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to -1)
        System.out.print(", ");
    }
    end();
  }
  public static void print(short[] a) {
      print(a, 0, a.length);
  }
  public static void
  print(String msg, short[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(short[] a, int from, int to) {
    start(from, to, a.length);
```

```
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to - 1)
        System.out.print(", ");
    }
    end();
  }
  public static void print(int[] a) {
      print(a, 0, a.length);
  }
  public static void
  print(String msg, int[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(int[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to - 1)
        System.out.print(", ");
    }
    end();
  }
  public static void print(long[] a) {
    print(a, 0, a.length);
  }
  public static void
  print(String msg, long[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(long[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to - 1)
        System.out.print(", ");
    }
```

```
    end();
  }
  public static void print(float[] a) {
      print(a, 0, a.length);
  }
  public static void
  print(String msg, float[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(float[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to - 1)
        System.out.print(", ");
    }
    end();
  }
  public static void print(double[] a) {
      print(a, 0, a.length);
  }
  public static void
  print(String msg, double[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
  }
  public static void
  print(double[] a, int from, int to){
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
      System.out.print(a[i]);
      if(i < to - 1)
        System.out.print(", ");
    }
    end();
  }
  // Fill an array using a generator:
  public static void
  fill(Object[] a, Generator gen) {
```

```
      fill(a, 0, a.length, gen);
  }
  public static void
  fill(Object[] a, int from, int to,
       Generator gen){
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void
  fill(boolean[] a, BooleanGenerator gen) {
      fill(a, 0, a.length, gen);
  }
  public static void
  fill(boolean[] a, int from, int to,
       BooleanGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void
  fill(byte[] a, ByteGenerator gen) {
      fill(a, 0, a.length, gen);
  }
  public static void
  fill(byte[] a, int from, int to,
       ByteGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void
  fill(char[] a, CharGenerator gen) {
      fill(a, 0, a.length, gen);
  }
  public static void
  fill(char[] a, int from, int to,
       CharGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void
  fill(short[] a, ShortGenerator gen) {
      fill(a, 0, a.length, gen);
```

```
  }
  public static void
  fill(short[] a, int from, int to,
       ShortGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void
  fill(int[] a, IntGenerator gen) {
      fill(a, 0, a.length, gen);
  }
  public static void
  fill(int[] a, int from, int to,
       IntGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void
  fill(long[] a, LongGenerator gen) {
      fill(a, 0, a.length, gen);
  }
  public static void
  fill(long[] a, int from, int to,
       LongGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void
  fill(float[] a, FloatGenerator gen) {
      fill(a, 0, a.length, gen);
  }
  public static void
  fill(float[] a, int from, int to,
       FloatGenerator gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next();
  }
  public static void
  fill(double[] a, DoubleGenerator gen) {
      fill(a, 0, a.length, gen);
  }
```

```
public static void
fill(double[] a, int from, int to,
     DoubleGenerator gen){
  for(int i = from; i < to; i++)
    a[i] = gen.next();
}
private static Random r = new Random();
public static class RandBooleanGenerator
implements BooleanGenerator {
  public boolean next()
    return r.nextBoolean();
  }
}
public static class RandByteGenerator
implements ByteGenerator {
  public byte next()
    return (byte)r.nextInt();
  }
}
static String ssource =
  "ABCDEFGHIJKLMNOPQRSTUVWXYZ" +
  "abcdefghijklmnopqrstuvwxyz";
static char[] src = ssource.toCharArray();
public static class RandCharGenerator
implements CharGenerator {
  public char next() {
    int pos = Math.abs(r.nextInt());
    return src[pos % src.length];
  }
}
public static class RandStringGenerator
implements Generator {
  private int len;
  private RandCharGenerator cg =
    new RandCharGenerator();
  public RandStringGenerator(int length) {
    len = length;
  }
  public Object next() {
    char[] buf = new char[len];
    for(int i = 0; i < len; i++)
```

```
        buf[i] = cg.next();
      return new String(buf);
    }
  }
  public static class RandShortGenerator
  implements ShortGenerator {
    public short next()
      return (short)r.nextInt();
    }
  }
  public static class RandIntGenerator
  implements IntGenerator {
    private int mod = 10000;
    public RandIntGenerator() {}
    public RandIntGenerator(int modulo) {
      mod = modulo;
    }
    public int next()
      return r.nextInt() % mod;
    }
  }
  public static class RandLongGenerator
  implements LongGenerator {
    public long next() { return r.nextLong(); }
  }
  public static class RandFloatGenerator
  implements FloatGenerator {
    public float next() { return r.nextFloat(); }
  }
  public static class RandDoubleGenerator
  implements DoubleGenerator {
    public double next() {return r.nextDouble();}
  }
} ///:~
```

爲了使用產生器（generator）來填補 array 內的元素空間，**fill()** 接受一個 reference，此 reference 指向某個適當的 generator **interface**，其中擁有 **next()**，能夠以某種方式產生正確型別的物件（這和 interface 的實作版本有關）。**fill()** 只是不斷呼叫 **next()**，直至填滿所欲填滿的範圍爲止。於

是，你便可以實作適當的 interface 而產生任何型式的產生器，並搭配 **fill()** 使用。

隨機產生器（random data generator）在測試上極爲有用，所以我寫了一組 inner classes，將所有基本型別產生器的 interface 實作出來，同時也實作了一個 **String** 產生器，用以表現 **Object** 的情況。你可以看到，**RandStringGenerator** 使用 **RandCharGenerator** 填入 **char** array，再將此 array 轉換爲 **String**。array 的容量由建構式引數決定。

爲了使產生的數字不至於過大，**RandIntGenerator** 會將產生的值對 10,000 取餘數。不過，重載版的建構式讓你可以選用更小的值。

以下程式用來測試整個程式庫，它同時也示範了這個程式庫的使用方式：

```
//: c09:TestArrays2.java
// Test and demonstrate Arrays2 utilities
import com.bruceeckel.util.*;

public class TestArrays2 {
  public static void main(String[] args) {
    int size = 6;
    // Or get the size from the command line:
    if(args.length != 0)
      size = Integer.parseInt(args[0]);
    boolean[] a1 = new boolean[size];
    byte[] a2 = new byte[size];
    char[] a3 = new char[size];
    short[] a4 = new short[size];
    int[] a5 = new int[size];
    long[] a6 = new long[size];
    float[] a7 = new float[size];
    double[] a8 = new double[size];
    String[] a9 = new String[size];
    Arrays2.fill(a1,
      new Arrays2.RandBooleanGenerator());
    Arrays2.print(a1);
    Arrays2.print("a1 = ", a1);
    Arrays2.print(a1, size/3, size/3 + size/3);
    Arrays2.fill(a2,
```

```
      new Arrays2.RandByteGenerator());
    Arrays2.print(a2);
    Arrays2.print("a2 = ", a2);
    Arrays2.print(a2, size/3, size/3 + size/3);
    Arrays2.fill(a3,
      new Arrays2.RandCharGenerator());
    Arrays2.print(a3);
    Arrays2.print("a3 = ", a3);
    Arrays2.print(a3, size/3, size/3 + size/3);
    Arrays2.fill(a4,
      new Arrays2.RandShortGenerator());
    Arrays2.print(a4);
    Arrays2.print("a4 = ", a4);
    Arrays2.print(a4, size/3, size/3 + size/3);
    Arrays2.fill(a5,
      new Arrays2.RandIntGenerator());
    Arrays2.print(a5);
    Arrays2.print("a5 = ", a5);
    Arrays2.print(a5, size/3, size/3 + size/3);
    Arrays2.fill(a6,
      new Arrays2.RandLongGenerator());
    Arrays2.print(a6);
    Arrays2.print("a6 = ", a6);
    Arrays2.print(a6, size/3, size/3 + size/3);
    Arrays2.fill(a7,
      new Arrays2.RandFloatGenerator());
    Arrays2.print(a7);
    Arrays2.print("a7 = ", a7);
    Arrays2.print(a7, size/3, size/3 + size/3);
    Arrays2.fill(a8,
      new Arrays2.RandDoubleGenerator());
    Arrays2.print(a8);
    Arrays2.print("a8 = ", a8);
    Arrays2.print(a8, size/3, size/3 + size/3);
    Arrays2.fill(a9,
      new Arrays2.RandStringGenerator(7));
    Arrays2.print(a9);
    Arrays2.print("a9 = ", a9);
    Arrays2.print(a9, size/3, size/3 + size/3);
  }
} ///:~
```

size 參數有個預設值，但你也可以在命令列（command line）中設定它。

array 的充填（filling）

Java 標準程式庫中的 **Arrayes** 也有個 **fill()**，但是它更爲普通，只是將單一數值（如果面對物件的話，則是將同一個 reference）複製到每個位置。實際用一下 **Array2.print()** 便能輕易體會 **Arrays.fill()** 的行爲模式：

```
//: c09:FillingArrays.java
// Using Arrays.fill()
import com.bruceeckel.util.*;
import java.util.*;

public class FillingArrays {
  public static void main(String[] args) {
    int size = 6;
    // Or get the size from the command line:
    if(args.length != 0)
      size = Integer.parseInt(args[0]);
    boolean[] a1 = new boolean[size];
    byte[] a2 = new byte[size];
    char[] a3 = new char[size];
    short[] a4 = new short[size];
    int[] a5 = new int[size];
    long[] a6 = new long[size];
    float[] a7 = new float[size];
    double[] a8 = new double[size];
    String[] a9 = new String[size];
    Arrays.fill(a1, true);
    Arrays2.print("a1 = ", a1);
    Arrays.fill(a2, (byte)11);
    Arrays2.print("a2 = ", a2);
    Arrays.fill(a3, 'x');
    Arrays2.print("a3 = ", a3);
    Arrays.fill(a4, (short)17);
    Arrays2.print("a4 = ", a4);
    Arrays.fill(a5, 19);
    Arrays2.print("a5 = ", a5);
    Arrays.fill(a6, 23);
```

```
    Arrays2.print("a6 = ", a6);
    Arrays.fill(a7, 29);
    Arrays2.print("a7 = ", a7);
    Arrays.fill(a8, 47);
    Arrays2.print("a8 = ", a8);
    Arrays.fill(a9, "Hello");
    Arrays2.print("a9 = ", a9);
    // Manipulating ranges:
    Arrays.fill(a9, 3, 5, "World");
    Arrays2.print("a9 = ", a9);
  }
} ///:~
```

你可以填滿整個 array，或是如最末兩行所展示的那樣，只填充一段範圍。但如果使用 **Arrays.fill()**，只能給定單一值。**Arrays2.fill()** 能產生更有趣的結果。

array 的複製

Java 標準程式庫提供一個名爲 **System.arraycopy()** 的 **static** 函式。和「自己手動利用 **for** 迴圈來執行複製」相比，這個函式提供更快速的 array 複製能力。**System.arraycopy()** 被重載以處理所有型別。以下便是處理 **int** array 的例子：

```
//: c09:CopyingArrays.java
// Using System.arraycopy()
import com.bruceeckel.util.*;
import java.util.*;

public class CopyingArrays {
  public static void main(String[] args) {
    int[] i = new int[25];
    int[] j = new int[25];
    Arrays.fill(i, 47);
    Arrays.fill(j, 99);
    Arrays2.print("i = ", i);
    Arrays2.print("j = ", j);
    System.arraycopy(i, 0, j, 0, i.length);
    Arrays2.print("j = ", j);
```

```
    int[] k = new int[10];
    Arrays.fill(k, 103);
    System.arraycopy(i, 0, k, 0, k.length);
    Arrays2.print("k = ", k);
    Arrays.fill(k, 103);
    System.arraycopy(k, 0, i, 0, k.length);
    Arrays2.print("i = ", i);
    // Objects:
    Integer[] u = new Integer[10];
    Integer[] v = new Integer[5];
    Arrays.fill(u, new Integer(47));
    Arrays.fill(v, new Integer(99));
    Arrays2.print("u = ", u);
    Arrays2.print("v = ", v);
    System.arraycopy(v, 0,
      u, u.length/2, v.length);
    Arrays2.print("u = ", u);
  }
} ///:~
```

傳入 **arraycopy()** 的引數包括來源端 array、來源端複製起點（偏移位置）、目的端 array、目的端接受起點（偏移位置），以及複製個數。當然，萬一發生 array 越界存取，便會引發異常（exceptions）。

這個例子展示一個事實：*privitives array* 和 *objects array* 都可被複製。不過，複製 *objects array* 時僅有 references 會被複製，元素（物件）本身並不會被複製。此即所謂淺層拷貝（*shallow copy*），請參考附錄 A。

arrays 的比較

Arrays 提供重載版的 **equals()**，藉以比較兩個 arrays 是否相等。它能夠處理 **Object** 和所有基本型別。兩個 arrays 必須擁有相同個數的元素，而且所有對應元素必須兩兩相等（使用元素自己的 **equals()** 來檢驗它們是否相等。如果元素屬於基本型別，則使用外覆類別的 **equals()** 來進行比較。如果面對的是 **int**，就使用 **Integer.equals()**。以下便是一例：

```
//: c09:ComparingArrays.java
// Using Arrays.equals()
```

```
import java.util.*;

public class ComparingArrays {
  public static void main(String[] args) {
    int[] a1 = new int[10];
    int[] a2 = new int[10];
    Arrays.fill(a1, 47);
    Arrays.fill(a2, 47);
    System.out.println(Arrays.equals(a1, a2));
    a2[3] = 11;
    System.out.println(Arrays.equals(a1, a2));
    String[] s1 = new String[5];
    Arrays.fill(s1, "Hi");
    String[] s2 = {"Hi", "Hi", "Hi", "Hi", "Hi"};
    System.out.println(Arrays.equals(s1, s2));
  }
} ///:~
```

一開始 **a1** 和 **a2** 完全相等，所以其輸出結果爲 "true"。接下來有一個元素被改變了，所以第二個輸出結果爲 "false"。最後一個情形，**s1** 的所有元素都指向同一物件，**s2** 卻擁有五個不同物件，不過由於 array 的相等測試是依據其內容來決定（透過 **Object.equals()**），所以結果爲 "true"。

array 元素的比較

Java 1.0 和 1.1 的程式庫缺乏演算法式的操作（algorithmic operations），連簡單的排序都沒有。標準程式庫應該具備哪些功能，實乃見仁見智，但上述情況對某些人來說無疑十分困惑。還好 Java 2 補救了這個問題，至少補強了排序能力。

撰寫一份泛型排序碼，面臨的問題是，排序必須依據物件實際型別來進行比較。當然，爲各種不同型別都分別撰寫一份排序用的函式，也是一種解決辦法，但你應該能夠理解，這種作法無法輕易用於新型別身上。

程式設計的主要目標，是要將「變動的事物和不變的事物隔離開來」。在這裡，保持不變的，就是泛用排序演算法，會變動的，則是物件的比較法。所以，如果不希望將物件比較動作寫死於許多不同的排序函式中，我

們可以採用 callback（回呼）技巧。透過 callback，「因勢而異的程式碼」可被封裝於自己的 class 內，而「永不變動的程式碼」則回頭呼叫前者。透過這個手法，便可讓不同的物件各自表述其不同的比較法，並將該比較法餵進同一個排序法則中。

Java 2 提供兩種比較機能。第一個是透過所謂 *natural comparison method*（自然比較法），藉由實作 **java.lang.Comparable** interface，使某個 class 具有比較能力。上述 interface 是個極爲簡單的 interface，只有一個函式：**compareTo()**，接受另一個 **Object** 做爲引數：引數小於自己時回傳負值，二者相等時回傳零值，引數大於自己時回傳正值。

以下就是一個實作了 **Comparable** 的 class，並使用 Java 標準程式庫中的 **Arrays.sort()** 示範比較動作。

```
//: c09:CompType.java
// Implementing Comparable in a class.
import com.bruceeckel.util.*;
import java.util.*;

public class CompType implements Comparable {
  int i;
  int j;
  public CompType(int n1, int n2)
    i = n1;
    j = n2;
  }
  public String toString()
    return "[i = " + i + ", j = " + j + "]";
  }
  public int compareTo(Object rv) {
    int rvi = ((CompType)rv).i;
    return (i < rvi ? -1 : (i == rvi ? 0 : 1));
  }
  private static Random r = new Random();
  private static int randInt() {
    return Math.abs(r.nextInt()) % 100;
  }
  public static Generator generator() {
    return new Generator() {
```

```
      public Object next() {
        return new CompType(randInt(),randInt());
      }
    };
  }
  public static void main(String[] args) {
    CompType[] a = new CompType[10];
    Arrays2.fill(a, generator());
    Arrays2.print("before sorting, a = ", a);
    Arrays.sort(a);
    Arrays2.print("after sorting, a = ", a);
  }
} ///:~
```

一旦你定義了比較函式，你便得負責決定你自己的物件與其他物件之間的比較意義爲何。在這個例子中，僅使用 **i** 值來進行比較，**j** 值被忽略。

static randInt() 會產生介於 0 和 100 之間的正值，**generator()** 則透過匿名的 inner class 產生出實作 **Generator** interface 的一個物件（請參考第 8 章）。這於是產生出 **CompType** 物件，並以亂數值加以初始化。**main()** 內利用 *generator* 充填 **CompType** array，再加以排序。呼叫 **sort()** 時，如果 **CompType** 並未實作出 **Comparable**，你便會收到編譯期錯誤訊息。

現在，假設某人交給你一個 class，此 class 並未實作 **Comparable**。或者交給你一個實作有 **Comparable** 的 class，但你不喜歡那種實作方式，也不想讓每個型別都擁有個別的比較函式。那麼你可以使用第二種物件比較方案，也就是撰寫一個 class，令它實作 **Comparator** interface。這個 interface 擁有兩個函式：**compare()** 和 **equals()**。但除非爲了特殊效率考量，否則不需要實作 **equals()**。因爲你所撰寫的所有 classes 會自動繼承 **Object**，而後者已經具備 **equals()**。所以你可以直接使用預設的 **Object euqlas()**，這樣便能滿足前述 interface 的規範。

Collections class（本章稍後我將會討論）內含唯一一個 **Comparator**，可將正常的排列順序顛倒過來。它可輕易被應用於 **CompType**：

```
//: c09:Reverse.java
// The Collecions.reverseOrder() Comparator.
import com.bruceeckel.util.*;
import java.util.*;

public class Reverse {
  public static void main(String[] args) {
    CompType[] a = new CompType[10];
    Arrays2.fill(a, CompType.generator());
    Arrays2.print("before sorting, a = ", a);
    Arrays.sort(a, Collections.reverseOrder());
    Arrays2.print("after sorting, a = ", a);
  }
} ///:~
```

呼叫 **Collections.reverseOrder()** 後你會得到一個 reference，指向 **Comparator**。

第二個例子是，以下的 **Comparator** 會依據 **CompType** 物件的 **j** 值（而非 **i** 值）來進行比較。

```
//: c09:ComparatorTest.java
// Implementing a Comparator for a class.
import com.bruceeckel.util.*;
import java.util.*;

class CompTypeComparator implements Comparator {
  public int compare(Object o1, Object o2) {
    int j1 = ((CompType)o1).j;
    int j2 = ((CompType)o2).j;
    return (j1 < j2 ? -1 : (j1 == j2 ? 0 : 1));
  }
}

public class ComparatorTest {
```

```
  public static void main(String[] args) {
    CompType[] a = new CompType[10];
    Arrays2.fill(a, CompType.generator());
    Arrays2.print("before sorting, a = ", a);
    Arrays.sort(a, new CompTypeComparator());
    Arrays2.print("after sorting, a = ", a);
  }
} ///:~
```

compare() 的第一引數小於、等於、或大於第二引數時，必須分別回傳負整數、零值、正整數。

array 的排序

使用內建的 sorting 函式時，你可以針對任何 *primitives array* 進行排序，也可以針對任何 *objects array* 進行排序（只要那些物件實作了 **Comparable** 或擁有相關之 **Comparator**）。這填補了 Java 程式庫的一個大漏洞 — 不論你相信與否，Java 1.0 和 1.1 竟然完全沒有支援 **String**s 的排序！以下這個範例會隨機產生 **String** 物件，並加排序：

```
//: c09:StringSorting.java
// Sorting an array of Strings.
import com.bruceeckel.util.*;
import java.util.*;

public class StringSorting {
  public static void main(String[] args) {
    String[] sa = new String[30];
    Arrays2.fill(sa,
      new Arrays2.RandStringGenerator(5));
    Arrays2.print("Before sorting: ", sa);
    Arrays.sort(sa);
    Arrays2.print("After sorting: ", sa);
  }
} ///:~
```

從 **String** 排序演算法的輸出結果中你會注意到，這個演算法乃根據「字典順序」進行排序。所以它會將所有以大寫字母為首的字詞置於以小寫字母為首的字詞之前（電話簿通常便是以此種方式排列）。你也可能會想要以

不分大小寫的方式排序，爲此你可定義一個 **Comparator** class，並覆寫 **String Comparable** 的預設行爲。基於重複運用的理由，這個 class 被加至 "util" package 中：

```
//: com:bruceeckel:util:AlphabeticComparator.java
// Keeping upper and lowercase letters together.
package com.bruceeckel.util;
import java.util.*;

public class AlphabeticComparator
implements Comparator{
  public int compare(Object o1, Object o2) {
    String s1 = (String)o1;
    String s2 = (String)o2;
    return s1.toLowerCase().compareTo(
      s2.toLowerCase());
  }
} ///:~
```

進行比較之前每個 **String** 都先被轉爲小寫。**String** 內建的 **compareTo()** 提供了我們想要的功能。

以下使用 **AlphabeticComparator** 進行測試：

```
//: c09:AlphabeticSorting.java
// Keeping upper and lowercase letters together.
import com.bruceeckel.util.*;
import java.util.*;

public class AlphabeticSorting {
  public static void main(String[] args) {
    String[] sa = new String[30];
    Arrays2.fill(sa,
      new Arrays2.RandStringGenerator(5));
    Arrays2.print("Before sorting: ", sa);
    Arrays.sort(sa, new AlphabeticComparator());
    Arrays2.print("After sorting: ", sa);
  }
} ///:~
```

Java 標準程式庫的排序演算法，會針對你所排序的型別來進行最佳化。面對基本型別時用的是 Quicksort；面對物件則採用 stable merge sort。所以你不應該浪費時間於其執行效率的改善上面，除非你的效能量測工具指出，你的排序動作的確是執行效率上的瓶頸。

在已排序的 array 中進行搜尋

array 排序完畢後，你可以使用 **Arrays.binarySearch()** 快速搜尋某個元素。但是千萬別在未經排序的 array 身上使用 **binarySearch()**，否則結果完全無法預測。下面這個例子先使用 **RandIntGenerator** 將數值填入 array，然後產生搜尋目標：

```
//: c09:ArraySearching.java
// Using Arrays.binarySearch().
import com.bruceeckel.util.*;
import java.util.*;

public class ArraySearching {
  public static void main(String[] args) {
    int[] a = new int[100];
    Arrays2.RandIntGenerator gen =
      new Arrays2.RandIntGenerator(1000);
    Arrays2.fill(a, gen);
    Arrays.sort(a);
    Arrays2.print("Sorted array: ", a);
    while(true) {
      int r = gen.next();
      int location = Arrays.binarySearch(a, r);
      if(location >= 0) {
        System.out.println("Location of " + r +
          " is " + location + ", a[" +
          location + "] = " + a[location]);
        break; // Out of while loop
      }
    }
  }
} ///:~
```

在 **while** 迴圈中，隨機值會被不斷產生出來，做爲搜尋目標，直到在 array 中確實找到該值爲止。

如果找到了搜尋目標，**Arrays.binarySearch()** 會回傳一個大於或等於零的值。反之則回傳一個負值，表示該值應被安插的位置 – 亦即如果希望該值加入 array 之後仍能保持 array 的排序狀態，該值應該被安插的位置。這個負值是：

```
-(insertion point) - 1
```

其中 insertion point 就是第一個大於搜尋值（或說鍵值，key）的元素的索引值。如果 array 之內所有元素都小於搜尋值，insertion point 即爲 **a.size()**。

如果 array 內含多個相同元素，上述搜尋動作不保證會搜出其中哪一個元素。這個演算法設計時並未考量「元素重複」的情況，只不過畢竟還能得兼。如果你需要的是一個無重複元素的 sorted list，請採用稍後即將介紹的 **TreeSet**，它會自動爲你處理所有細節。只有當 **TreeSet** 成爲效率瓶頸時，你才需要以一個手動維護的 array 來取代 **TreeSet**。

如果你已經以 **Comparator** 對 *objects array* 排序（注意，*primitives array* 不允許以 **Comparator** 排序），那麼當你執行 **binarySearch()** 時（用的是其重載版本），就得使用同一個 **Comparator**。例如我們修改 **AlphabeticSorting.java** 程式，執行搜尋動作：

```
//: c09:AlphabeticSearch.java
// Searching with a Comparator.
import com.bruceeckel.util.*;
import java.util.*;

public class AlphabeticSearch {
  public static void main(String[] args) {
    String[] sa = new String[30];
    Arrays2.fill(sa,
      new Arrays2.RandStringGenerator(5));
    AlphabeticComparator comp =
      new AlphabeticComparator();
    Arrays.sort(sa, comp);
    int index =
```

```
      Arrays.binarySearch(sa, sa[10], comp);
    System.out.println("Index = " + index);
  }
} ///:~
```

Comparator 必須被傳入 **binarySearch()** 重載版本，做爲第三引數。上例的搜尋一定會成功，因爲搜尋目標是由 array 自身取出的。

Array 總結

總結至今所討論的內容，當你持有一大群物件，你的第一（也是最高效能）的選擇應該是 array。如果你想持有一群基本型別數值，也只能選擇 array。本章剩餘篇幅中，我會探討更一般化的情形。在那些情況下，撰寫程式時你並不知道究竟需要多少物件，因此你需要更複雜的方法來持有它們。Java 提供一套容器類別程式庫（container classes library）用以解決這個問題，其基本類型包括：**List**、**Set**、**Map**。你可以運用這些工具來解決極多問題。

這些容器各自有其特性。例如 **Set** 針對每個值只會儲存一個物件；**Map** 是個關聯式 array（*associative array*），讓你得以將某個物件關聯至另一個物件。Java 容器會自動調整自己的容量，所以，和 array 不同，你可以置入任意個數的物件，撰寫應用程式時不需擔心容量問題。

容器（containers）簡介

對我而言，容器用於一個嶄新的專案開發上，是最有威力的工具之一，因爲它可以大幅增加你的編程實力。Java 2 容器是針對 Java 1.0 和 1.1 中頗爲差勁的表現所做的徹底重新設計[4]。重新設計後的結果，某些事物更嚴謹也更實用，並且豐富了容器程式庫（containers library）的機能，提供了 linked-list、queue、deques（兩端開口的 queue，讀做 "decks"）。

[4] 由 Sun 公司的 Joshua Bloch 操刀。

容器程式庫的設計十分困難（大部份程式庫都如此）。在 C++ 中，容器類別以許多不同的 classes 呈現。這比過去什麼都沒有的情況當然好多了。但這種方式並未轉移到 Java 身上。我曾經看過一個極端的例子，某個市售容器庫只有單一一個 "container" class，其行爲模式既像個線性序列（linear sequence）又像個關聯式（associative）array。Java 2 容器庫取乎中庸：你認爲一個成熟的容器庫該有的功能它都具備了，但比 C++容器類別及其他容器庫更易學習和運用。這樣的結果似乎有些詭異，但由於它和早先的 Java 程式庫的某些決策不同，因此這種詭異並非偶然，是權衡複雜度之後的謹慎考量結果。你也許得花點時間才能夠上手，但我想你很快會發現，自己將快速學成並愛上這些新工具。

Java 2 容器庫所解決的是物件持有問題，並將它區分爲兩個不同的概念：

1. **Collection**：一組各自獨立的元素，通常擁有相同的套用規則：**List** 必須以特定次序來持有各元素；**Set** 無法擁有重複元素 — *bag* 並無如此限制（Java 容器庫未實作有 bag，因爲 **List**s 已提供足夠功能）。

2. **Map**：一群成對的 *key-value* 物件（譯註：由「鍵值/實值」成對構成的物件）。乍見之下它似乎應該是一個由 "pair" 物件組成的 **Collection**，但是當你試著以這種方法實現時，整個設計會變得極爲笨拙。使其成爲一個獨立概念反倒更清楚些。另一方面，如果能夠產生 **Collection** 來表示 **Map** 的內容，將會極爲便利，因此 **Map** 可以回傳一個由 *key*（鍵值）形成的 **Set**，或是一個由 *value*（實值）形成的 **Collection**，或是一個內含 *key-value* pairs 的 **Set**。**Map**s 就和 array 一樣，無需加入新概念，便可輕易擴展成多維形式：只要讓 **Map** 的實值（*values*）又是個 **Map** 即可，後者的實值（*values*）還可以再是 **Map**s…依此類推。

我首先探討容器的一般性質，然後再深入研究其細節，最後我們看看爲什麼需要某些容器的特別版本，以及它們的選擇方式。

容器的列印

和 array 不同的是，無需任何額外處理，便能巧妙印出容器內容。以下便是一例，此例同時爲你介紹基本的容器類型：

```
//: c09:PrintingContainers.java
// Containers print themselves automatically.
import java.util.*;

public class PrintingContainers {
  static Collection fill(Collection c) {
    c.add("dog");
    c.add("dog");
    c.add("cat");
    return c;
  }
  static Map fill(Map m) {
    m.put("dog", "Bosco");
    m.put("dog", "Spot");
    m.put("cat", "Rags");
    return m;
  }
  public static void main(String[] args) {
    System.out.println(fill(new ArrayList()));
    System.out.println(fill(new HashSet()));
    System.out.println(fill(new HashMap()));
  }
} ///:~
```

一如先前所述，在 Java 容器庫中有兩個基本分類。其間的區別主要在於容器內每個位置所儲存的元素個數。屬於 **Collection** 類型者，其內的每個位置僅持有一個元素（**Collection** 這個名稱可能會造成若干誤會，因爲整個 Java 容器庫往往又被稱爲 "collection"）。這一類型包括有：**List**，以特定次序儲存一組元素；**Set**，元素不得重複。**ArrayList** 是一種 **List**，而 **HashSet** 則是一種 **Set**。**add()** 可將元素加入任何一種 **Collection**。

Map 所持有的則是 *key-value* pairs，像個小型資料庫。上述實例運用了 Map 中的一種：**HashMap**。如果你擁有某個 **Map**，它將美國州名關聯至

其首府名稱，而你想知道俄亥俄州的首府名稱，那麼，搜尋方式就像對 array 進行索引動作一樣。所以 **Map**s 也被稱爲關聯式（*associative*）*arrays*。欲將元素加至 **Map**，可使用 **put()**，它接收 *key* 和 *value* 做爲引數。上例只示範元素的加入方式，但沒有在加入之後執行搜尋動作。稍後會說明搜尋方式。

fill() 的重載版本可用來充填 **Collection**s 和 **Map**s。如果你檢視輸出結果，你會發現，預設的列印功能（由各容器本身的 **toString()** 提供）所產生的結果極具可讀性，所以我們不必像面對 array 那樣地對列印提供額外支援：

```
[dog, dog, cat]
[cat, dog]
{cat=Rags, dog=Spot}
```

Collection 的列印結果以方括號括住，每個元素之間以逗號相隔。**Map** 的列印結果則以大括號括住，*key* 和 *value* 之間以等號相接，*key* 在左側，*value* 在右側。

你馬上可以觀察到不同容器的基本行爲。**List** 會以元素安插次序來放置元素，不會重新排列或編修。**Set** 不接受重複元素，它會使用自己內部的一個排列（ordering）機制。如果你只關心某個物件是否存在，而不關心它們的出現順序，那麼你應該使用 **List**。**Map** 也不接受重複元素，重複與否乃以鍵值（*key*）判斷。**Map** 也擁有自己的內部排列（ordering）機制，它一點也不在意你安插元素時的次序。

容器的充填

雖然容器的列印問題已獲解決，但是容器的充填動作卻和先前討論的 **java.util.Arrays** 一樣不足。和 **Arrays** 一樣，有個相應的 **Collections** class，含有一些 **static** 函式，其中之一便稱爲 **fill()**。它只是將同一個 object reference 複製到容器的每個位置上，而且只對 **List** 有效，無法作用於 **Set**s 或 **Map**s：

```java
//: c09:FillingLists.java
// The Collections.fill() method.
import java.util.*;

public class FillingLists {
  public static void main(String[] args) {
    List list = new ArrayList();
    for(int i = 0; i < 10; i++)
      list.add("");
    Collections.fill(list, "Hello");
    System.out.println(list);
  }
} ///:~
```

這個函式只能替換掉原先已存在於 **List** 中的元素，無法加入新元素。這使它顯得更是沒有用處。

爲了撰寫更有趣的例子，這裡我提供一個用來進行補強的 **Collection2** 程式庫（是的，**com.bruceeckel.util** 之中有些東西純粹是爲了更方便），其中的 **fill()** 會利用 generator（自動產生器）來加入元素，並允許你指定想要加入的元素個數。先前所定義的 **Generator interface** 同樣能用於 **Collection**s 身上，但 **Map** 就得有自己的 generator **interface**，因爲每次呼叫 **next()** 都得產生成對的物件才行（其一爲 *key*，另一爲 *value*）。以下便是 **Pair** class：

```java
//: com:bruceeckel:util:Pair.java
package com.bruceeckel.util;
public class Pair {
  public Object key, value;
  Pair(Object k, Object v) {
    key = k;
    value = v;
  }
} ///:~
```

下面是 **Pair** 的 generator **interface**：

```
//: com:bruceeckel:util:MapGenerator.java
package com.bruceeckel.util;
public interface MapGenerator {
  Pair next();
} ///:~
```

有了這些 classes，我們便可以發展一套處理容器類別的公用程式：

```
//: com:bruceeckel:util:Collections2.java
// To fill any type of container
// using a generator object.
package com.bruceeckel.util;
import java.util.*;

public class Collections2 {
  // Fill an array using a generator:
  public static void
  fill(Collection c, Generator gen, int count) {
    for(int i = 0; i < count; i++)
      c.add(gen.next());
  }
  public static void
  fill(Map m, MapGenerator gen, int count) {
    for(int i = 0; i < count; i++) {
      Pair p = gen.next();
      m.put(p.key, p.value);
    }
  }
  public static class RandStringPairGenerator
  implements MapGenerator {
    private Arrays2.RandStringGenerator gen;
    public RandStringPairGenerator(int len) {
      gen = new Arrays2.RandStringGenerator(len);
    }
    public Pair next() {
      return new Pair(gen.next(), gen.next());
    }
  }
  // Default object so you don't have
  // to create your own:
  public static RandStringPairGenerator rsp =
```

```
    new RandStringPairGenerator(10);
  public static class StringPairGenerator
  implements MapGenerator {
    private int index = -1;
    private String[][] d;
    public StringPairGenerator(String[][] data) {
      d = data;
    }
    public Pair next() {
      // Force the index to wrap:
      index = (index + 1) % d.length;
      return new Pair(d[index][0], d[index][1]);
    }
    public StringPairGenerator reset()
      index = -1;
      return this;
    }
  }
  // Use a predefined dataset:
  public static StringPairGenerator geography =
    new StringPairGenerator(
      CountryCapitals.pairs);
  // Produce a sequence from a 2D array:
  public static class StringGenerator
  implements Generator {
    private String[][] d;
    private int position;
    private int index = -1;
    public
    StringGenerator(String[][] data, int pos) {
      d = data;
      position = pos;
    }
    public Object next() {
      // Force the index to wrap:
      index = (index + 1) % d.length;
      return d[index][position];
    }
    public StringGenerator reset()
      index = -1;
      return this;
```

```
    }
  }
  // Use a predefined dataset:
  public static StringGenerator countries =
    new StringGenerator(CountryCapitals.pairs,0);
  public static StringGenerator capitals =
    new StringGenerator(CountryCapitals.pairs,1);
} ///:~
```

兩個版本的 **fill()** 都接受一個「決定容器元素個數」的引數。此外我爲 **Map** 提供兩個自動產生器：(1) **RandStringPairGenerator**，它會產生任意個數的、成對的、隨機的 **Strings**，字串長度由建構式引數決定；(2) **StringPairGenerator**，它能夠根據外界給予的二維 **String** array，產生成對的 **Strings**。**StringGenerator** 也接收二維 **String** array，但它產生的是單一元素而非一個 **Pairs**。**static rsp**, **geography**, **countries**, **capitals** 等物件各自提供了預製的產生器，後三者使用的是世界各國的名稱和首都。請注意，如果你嘗試產生更多成對資訊，超過了可用資料，產生器會自動回繞到啓始點。而當你將這些成對資訊置入 Map 時，重覆的內容會被自動略去。

以下便是預先定義好的資料集，其中含有各個國家的名稱和其首都。此份資料以較小字型顯示，以節省空間：

```
//: com:bruceeckel:util:CountryCapitals.java
package com.bruceeckel.util;
public class CountryCapitals {
  public static final String[][] pairs = {
    // Africa
    {"ALGERIA","Algiers"}, {"ANGOLA","Luanda"},
    {"BENIN","Porto-Novo"}, {"BOTSWANA","Gaberone"},
    {"BURKINA FASO","Ouagadougou"}, {"BURUNDI","Bujumbura"},
    {"CAMEROON","Yaounde"}, {"CAPE VERDE","Praia"},
    {"CENTRAL AFRICAN REPUBLIC","Bangui"},
    {"CHAD","N'djamena"},  {"COMOROS","Moroni"},
    {"CONGO","Brazzaville"}, {"DJIBOUTI","Dijibouti"},
    {"EGYPT","Cairo"}, {"EQUATORIAL GUINEA","Malabo"},
    {"ERITREA","Asmara"}, {"ETHIOPIA","Addis Ababa"},
    {"GABON","Libreville"}, {"THE GAMBIA","Banjul"},
    {"GHANA","Accra"}, {"GUINEA","Conakry"},
    {"GUINEA","-"}, {"BISSAU","Bissau"},
    {"CETE D'IVOIR (IVORY COAST)","Yamoussoukro"},
    {"KENYA","Nairobi"}, {"LESOTHO","Maseru"},
    {"LIBERIA","Monrovia"}, {"LIBYA","Tripoli"},
```

```
{"MADAGASCAR","Antananarivo"}, {"MALAWI","Lilongwe"},
{"MALI","Bamako"}, {"MAURITANIA","Nouakchott"},
{"MAURITIUS","Port Louis"}, {"MOROCCO","Rabat"},
{"MOZAMBIQUE","Maputo"}, {"NAMIBIA","Windhoek"},
{"NIGER","Niamey"}, {"NIGERIA","Abuja"},
{"RWANDA","Kigali"}, {"SAO TOME E PRINCIPE","Sao Tome"},
{"SENEGAL","Dakar"}, {"SEYCHELLES","Victoria"},
{"SIERRA LEONE","Freetown"}, {"SOMALIA","Mogadishu"},
{"SOUTH AFRICA","Pretoria/Cape Town"}, {"SUDAN","Khartoum"},
{"SWAZILAND","Mbabane"}, {"TANZANIA","Dodoma"},
{"TOGO","Lome"}, {"TUNISIA","Tunis"},
{"UGANDA","Kampala"},
{"DEMOCRATIC REPUBLIC OF THE CONGO (ZAIRE)","Kinshasa"},
{"ZAMBIA","Lusaka"}, {"ZIMBABWE","Harare"},
// Asia
{"AFGHANISTAN","Kabul"}, {"BAHRAIN","Manama"},
{"BANGLADESH","Dhaka"}, {"BHUTAN","Thimphu"},
{"BRUNEI","Bandar Seri Begawan"}, {"CAMBODIA","Phnom Penh"},
{"CHINA","Beijing"}, {"CYPRUS","Nicosia"},
{"INDIA","New Delhi"}, {"INDONESIA","Jakarta"},
{"IRAN","Tehran"}, {"IRAQ","Baghdad"},
{"ISRAEL","Jerusalem"}, {"JAPAN","Tokyo"},
{"JORDAN","Amman"}, {"KUWAIT","Kuwait City"},
{"LAOS","Vientiane"}, {"LEBANON","Beirut"},
{"MALAYSIA","Kuala Lumpur"}, {"THE MALDIVES","Male"},
{"MONGOLIA","Ulan Bator"}, {"MYANMAR (BURMA)","Rangoon"},
{"NEPAL","Katmandu"}, {"NORTH KOREA","P'yongyang"},
{"OMAN","Muscat"}, {"PAKISTAN","Islamabad"},
{"PHILIPPINES","Manila"}, {"QATAR","Doha"},
{"SAUDI ARABIA","Riyadh"}, {"SINGAPORE","Singapore"},
{"SOUTH KOREA","Seoul"}, {"SRI LANKA","Colombo"},
{"SYRIA","Damascus"}, {"TAIWAN (REPUBLIC OF CHINA)","Taipei"},
{"THAILAND","Bangkok"}, {"TURKEY","Ankara"},
{"UNITED ARAB EMIRATES","Abu Dhabi"}, {"VIETNAM","Hanoi"},
{"YEMEN","Sana'a"},
// Australia and Oceania
{"AUSTRALIA","Canberra"}, {"FIJI","Suva"},
{"KIRIBATI","Bairiki"},
{"MARSHALL ISLANDS","Dalap-Uliga-Darrit"},
{"MICRONESIA","Palikir"}, {"NAURU","Yaren"},
{"NEW ZEALAND","Wellington"}, {"PALAU","Koror"},
{"PAPUA NEW GUINEA","Port Moresby"},
{"SOLOMON ISLANDS","Honaira"}, {"TONGA","Nuku'alofa"},
{"TUVALU","Fongafale"}, {"VANUATU","< Port-Vila"},
{"WESTERN SAMOA","Apia"},
// Eastern Europe and former USSR
{"ARMENIA","Yerevan"}, {"AZERBAIJAN","Baku"},
{"BELARUS (BYELORUSSIA)","Minsk"}, {"GEORGIA","Tbilisi"},
{"KAZAKSTAN","Almaty"}, {"KYRGYZSTAN","Alma-Ata"},
{"MOLDOVA","Chisinau"}, {"RUSSIA","Moscow"},
{"TAJIKISTAN","Dushanbe"}, {"TURKMENISTAN","Ashkabad"},
{"UKRAINE","Kyiv"}, {"UZBEKISTAN","Tashkent"},
```

```
    // Europe
    {"ALBANIA","Tirana"}, {"ANDORRA","Andorra la Vella"},
    {"AUSTRIA","Vienna"}, {"BELGIUM","Brussels"},
    {"BOSNIA","-"}, {"HERZEGOVINA","Sarajevo"},
    {"CROATIA","Zagreb"}, {"CZECH REPUBLIC","Prague"},
    {"DENMARK","Copenhagen"}, {"ESTONIA","Tallinn"},
    {"FINLAND","Helsinki"}, {"FRANCE","Paris"},
    {"GERMANY","Berlin"}, {"GREECE","Athens"},
    {"HUNGARY","Budapest"}, {"ICELAND","Reykjavik"},
    {"IRELAND","Dublin"}, {"ITALY","Rome"},
    {"LATVIA","Riga"}, {"LIECHTENSTEIN","Vaduz"},
    {"LITHUANIA","Vilnius"}, {"LUXEMBOURG","Luxembourg"},
    {"MACEDONIA","Skopje"}, {"MALTA","Valletta"},
    {"MONACO","Monaco"}, {"MONTENEGRO","Podgorica"},
    {"THE NETHERLANDS","Amsterdam"}, {"NORWAY","Oslo"},
    {"POLAND","Warsaw"}, {"PORTUGAL","Lisbon"},
    {"ROMANIA","Bucharest"}, {"SAN MARINO","San Marino"},
    {"SERBIA","Belgrade"}, {"SLOVAKIA","Bratislava"},
    {"SLOVENIA","Ljujiana"}, {"SPAIN","Madrid"},
    {"SWEDEN","Stockholm"}, {"SWITZERLAND","Berne"},
    {"UNITED KINGDOM","London"}, {"VATICAN CITY","---"},
    // North and Central America
    {"ANTIGUA AND BARBUDA","Saint John's"}, {"BAHAMAS","Nassau"},
    {"BARBADOS","Bridgetown"}, {"BELIZE","Belmopan"},
    {"CANADA","Ottawa"}, {"COSTA RICA","San Jose"},
    {"CUBA","Havana"}, {"DOMINICA","Roseau"},
    {"DOMINICAN REPUBLIC","Santo Domingo"},
    {"EL SALVADOR","San Salvador"}, {"GRENADA","Saint George's"},
    {"GUATEMALA","Guatemala City"}, {"HAITI","Port-au-Prince"},
    {"HONDURAS","Tegucigalpa"}, {"JAMAICA","Kingston"},
    {"MEXICO","Mexico City"}, {"NICARAGUA","Managua"},
    {"PANAMA","Panama City"}, {"ST. KITTS","-"},
    {"NEVIS","Basseterre"}, {"ST. LUCIA","Castries"},
    {"ST. VINCENT AND THE GRENADINES","Kingstown"},
    {"UNITED STATES OF AMERICA","Washington, D.C."},
    // South America
    {"ARGENTINA","Buenos Aires"},
    {"BOLIVIA","Sucre (legal)/La Paz(administrative)"},
    {"BRAZIL","Brasilia"}, {"CHILE","Santiago"},
    {"COLOMBIA","Bogota"}, {"ECUADOR","Quito"},
    {"GUYANA","Georgetown"}, {"PARAGUAY","Asuncion"},
    {"PERU","Lima"}, {"SURINAME","Paramaribo"},
    {"TRINIDAD AND TOBAGO","Port of Spain"},
    {"URUGUAY","Montevideo"}, {"VENEZUELA","Caracas"},
  };
} ///:~
```

這些都只不過是二維的 **String** array[5]。下面是個簡單測試程式，示範如何使用 **fill()** 和自動產生器：

```
//: c09:FillTest.java
import com.bruceeckel.util.*;
import java.util.*;

public class FillTest {
  static Generator sg =
    new Arrays2.RandStringGenerator(7);
  public static void main(String[] args) {
    List list = new ArrayList();
    Collections2.fill(list, sg, 25);
    System.out.println(list + "\n");
    List list2 = new ArrayList();
    Collections2.fill(list2,
      Collections2.capitals, 25);
    System.out.println(list2 + "\n");
    Set set = new HashSet();
    Collections2.fill(set, sg, 25);
    System.out.println(set + "\n");
    Map m = new HashMap();
    Collections2.fill(m, Collections2.rsp, 25);
    System.out.println(m + "\n");
    Map m2 = new HashMap();
    Collections2.fill(m2,
      Collections2.geography, 25);
    System.out.println(m2);
  }
} ///:~
```

有了上述工具，你可以將各種有趣資料填入各個容器內，藉以輕鬆測試不同的容器。

---

[5] 這些資料是在 Internet 上找到，再以 Python 程式（請參考 *www.Python.org*）加以處理。

容器的缺點：元素型別未定

Java 容器的缺點是，一旦你將物件置於容器內，你便損失了他的型別資訊。這是因為當初撰寫容器類別的那些人，並不知道你置於容器內的元素型別可能為何，而且讓容器僅僅儲存某種特定型別，會使容器無法成為通用工具。容器所持有的其實是一個個 reference 指向 **Object**（**Object** 是所有 Java classes 的根源），進而才能儲存任意型別。當然這不包括基本型別，因為基本型別並不繼承自任何 classes。這是很棒的解決方法，但是：

1. 由於你將 object reference 置入容器時已捨棄（割除）型別資訊，所以容器對於其所容納的元素的型別，沒有任何限制。這麼說好了，即使你只打算讓你的容器儲存「貓」，其他人仍然可以輕易將「狗」置入這個容器內。

2. 由於型別資訊已失，所以容器唯一知道的事情就是，它所持有的乃是指向物件的一些 references。使用之前你必須先將元素轉為正確型別。

往好的一面說，Java 並不至於讓你誤用容器內的物件。如果你將「狗」置於「貓」容器內，然後企圖將容器內的所有事物都視為「貓」，那麼當你將「狗」（的 reference）自「貓」容器取出並試著將它轉型為「貓」時，便會得到執行期異常。

下面這個例子使用最基本、最常被使用的容器：**ArrayList**。初學者可以將 **ArrayList** 想像成一種「會自動擴增容量的 array」。**ArrayList** 的使用方式十分簡單：產生 **ArrayList**、利用 **add()** 將物件置入、利用 **get()** 配合索引值將它們取出。這一切就和 array 的使用方式完全相同，只不過少了中括號罷了[6]。

ArrayList 還有一個 **size()**，讓你得以知道目前的元素個數，如此一來你才不會不慎超過邊界而引發異常。

---

[6] 此處如果有運算子重載（operator overloading）的功能就太好了（譯註：C++有）。

現在，首先，我們設計 **Cat** class 和 **Dog** class：

```
//: c09:Cat.java
public class Cat {
  private int catNumber;
  Cat(int i) { catNumber = i; }
  void print() {
    System.out.println("Cat #" + catNumber);
  }
} ///:~

//: c09:Dog.java
public class Dog {
  private int dogNumber;
  Dog(int i) { dogNumber = i; }
  void print() {
    System.out.println("Dog #" + dogNumber);
  }
} ///:~
```

將 **Cat**s 和 **Dog**s 置入容器內，然後再將它們取出：

```
//: c09:CatsAndDogs.java
// Simple container example.
import java.util.*;

public class CatsAndDogs {
  public static void main(String[] args) {
    ArrayList cats = new ArrayList();
    for(int i = 0; i < 7; i++)
      cats.add(new Cat(i));
    // Not a problem to add a dog to cats:
    cats.add(new Dog(7));
    for(int i = 0; i < cats.size(); i++)
      ((Cat)cats.get(i)).print();
    // Dog is detected only at run-time
  }
} ///:~
```

Cat 和 **Dog** 並不相同，它們除了都是 **Object**s 之外（如果你未曾明確指出你所繼承的 class，便會自動繼承 **Object**），沒有其他任何共通特性。由

於 **ArrayList** 所持有的乃是 **Objects**，所以你不但可以使用 **ArrayList** 的 **add()** 將 **Cat** 物件置於這個容器內，也可以將 **Dog** 物件加入，不會發生編譯期錯誤乃至執行期錯誤。當你使用 **ArrayList** 的 **get()** 將「你以為是 **Cat** 物件」的東西取出時，你取得的是一個 object reference，它必須先被轉型為 **Cat** 才能使用。於是你得使用圓括號將整個算式括住，強迫轉型動作先於 **Cat print()** 之前發生，否則便會發生語法錯誤。而當你試著將 **Dog** 物件轉型為 **Cat** 時，你會收到一個執行期異常（exception）。

這可不只是個煩惱而已。這種情況可能產生難以查覺的程式臭蟲。如果程式某處（或多處）將物件置入容器，你會發現，只要程式某處收到一個異常而它代表「有個不正確的物件被置於容器內」，你就得找出究竟何處進行了這樣的錯誤置入動作。不過，就好的一面來說，能夠擁有標準程式庫來協助編程，畢竟是很方便的，姑且不論其中可能的不足和拙劣。

譯註：Java Spec. Request（JSR，Java 規格需求）第 014 號，要求 Java 容納泛型機制。下筆此刻最新的 JDK1.4 已採用 "Generic Java" 技術，成了名副其實的「泛型爪哇」，允許程式員在運用 Java 容器時，指定元素型別。語法非常近似於 C++，唯底層實作技術完全不同。

有時候也總是可以運作

事實上，某些情況下，即便沒有轉型至原先的型別，仍然可以運作無誤。有一種情況尤其特別：編譯器對 **String** class 提供了一些額外支援，使它可以平滑運作。當編譯器預期收到 **String** object 而卻沒有收到時，它會自動呼叫 **toString()** 函式，後者定義於 **Object** 之中，可被任何 Java classes 覆寫。這個 **toString()** 能夠產生編譯器想要的 **String** object 並被用於必要地點。

因此，如果想列印你自己的 class objects，只要覆寫其 **toString()** 函式即可，例如：

```
//: c09:Mouse.java
// Overriding toString().
public class Mouse {
  private int mouseNumber;
  Mouse(int i) { mouseNumber = i; }
  // Override Object.toString():
```

```
  public String toString() {
    return "This is Mouse #" + mouseNumber;
  }
  public int getNumber() {
    return mouseNumber;
  }
} ///:~

//: c09:WorksAnyway.java
// In special cases, things just
// seem to work correctly.
import java.util.*;

class MouseTrap {
  static void caughtYa(Object m) {
    Mouse mouse = (Mouse)m; // Cast from Object
    System.out.println("Mouse: " +
      mouse.getNumber());
  }
}

public class WorksAnyway {
  public static void main(String[] args) {
    ArrayList mice = new ArrayList();
    for(int i = 0; i < 3; i++)
      mice.add(new Mouse(i));
    for(int i = 0; i < mice.size(); i++) {
      // No cast necessary, automatic
      // call to Object.toString():
      System.out.println(
        "Free mouse: " + mice.get(i));
      MouseTrap.caughtYa(mice.get(i));
    }
  }
} ///:~
```

你可以看到 **Mouse** 之內覆寫了 **toString()**。在 **main()** 的第二個 **for** 迴圈裡，你可以看到這行述句：

```
System.out.println("Free mouse: " + mice.get(i));
```

在 '+' 符號之後，編譯器預期見到一個 **String**，但 **get()** 回傳的是一個 **Object**。爲了得到所需的 **String**，編譯器會自動呼叫 **toString()**。不幸的是這種神奇效果只發生在 **String** 身上；其他型別無福享受。

第二種作法是將轉型動作隱藏於 **MouseTrap** 內。你看，**caughtYa()** 接收的並非是個 **Mouse** 而是一個 **Object**，之後才將 **Object** 轉型爲 **Mouse**。這種作法很魯莽，因爲接收的是 **Object**，所以任何東西都可被傳進來。不過，萬一轉型動作失敗（假設你傳入的型別錯誤的話）便會在執行期發生異常。這雖然不比執行期的檢查好，但仍然足夠穩當。請注意，使用以下函式時：

```
MouseTrap.caughtYa(mice.get(i));
```

並不需要進行任何轉型動作。

製作一個具有型別意識（type-conscious）的 ArrayList

你可能不想就此放棄前述議題。更嚴謹的解決方法就是根據 **ArrayList** 撰寫新的 class，使後者能夠只接收你所指定的型別，並且只回傳該型別：

```
//: c09:MouseList.java
// A type-conscious ArrayList.
import java.util.*;

public class MouseList {
  private ArrayList list = new ArrayList();
  public void add(Mouse m) {
    list.add(m);
  }
  public Mouse get(int index) {
    return (Mouse)list.get(index);
  }
  public int size() { return list.size(); }
} ///:~
```

以下便是這個新容器的測試：

```
//: c09:MouseListTest.java
```

```
public class MouseListTest {
  public static void main(String[] args) {
    MouseList mice = new MouseList();
    for(int i = 0; i < 3; i++)
      mice.add(new Mouse(i));
    for(int i = 0; i < mice.size(); i++)
      MouseTrap.caughtYa(mice.get(i));
  }
} ///:~
```

它和前一個例子十分相像，只不過新的 **MouseList** class 擁有一個型別為 **ArrayList** 的 **private** 成員，至於函式則和 **ArrayList** 一模一樣。但它並不接收、也不回傳泛化的 **Objects**，而是回傳 **Mouse** 物件。

請注意，如果 **MouseList** 改採「繼承 **ArrayList**」的方式，那麼 **add(Mouse)** 就只會重載既有的 **add(Object)**，而不會對「被加入的物件」的型別進行限制。於是 **MouseList** 變成 **ArrayList** 的一個代理人（*surrogate*），在轉遞工作之前先執行一些動作（請參考《*Thinking in Patterns with Java*》，可自 *www.BruceEckel.com* 下載）。

由於 **MouseList** 只接受 **Mouse**，所以如果程式寫成：

```
mice.add(new Pigeon());
```

你會在編譯期獲得錯誤訊息。這種方法雖然從撰碼觀點來看比較麻煩，但是當你使用了不正確的型別時，能夠立即回報。

請注意，使用 **get()** 時不需要任何轉型動作，因為獲得的永遠是 **Mouse**。

參數化型別（parameterized types）

這類問題並不罕見：許多情況下你得依據其他型別來產生新型別；如果能在編譯期擁有特定的型別資訊，將會非常有用。此即所謂「參數化型別」（*parameterized type*）觀念。C++程式語言以所謂 **templates**（模版）對此直接提供支援。Java 未來版本有可能支援某種參數化型別；目前居於領先地位的某些技術團隊，已有能力自動產生類似 **MouseList** 之類的 classes。（譯註：目前 JDK1.4+JSR14 已能支援「參數化型別」）

迭代器（Iterators）

任何容器類別都必須提供某種「物件置入方式」和「物件取出方式」。畢竟容器的主要職責所在便是：儲存（持有）物件。在 **ArrayList** 中，**add()** 是插入物件的方式，**get()** 是取出物件的方式。**ArrayList** 極具彈性，你可以在任意時間選擇任何一個物件，並且可以一次使用多個不同的索引來選擇多個元素。

但是如果你開始想以較高層次的方式來思考，便會發現一個缺點：你得知道容器的確切類型爲何，才能加以運用。一開始這可能不是壞事，但如果你一開始採用 **ArrayList**，後來發現改用 **LinkedList** 做爲容器更具效率，那麼又該如何？或者假設你想要撰寫一段泛型（generic）程式碼，這段程式碼不知道也不在意它所使用的容器類型究竟爲何，如此一來它就可被用於不同類型的容器身上，不需要每次重新撰寫一份。

所謂「迭代器（*iterator*）」，可被用來達成這種概念。迭代器是個物件，其職責便是走訪以及選擇序列（sequence）中的一連串物件。客端程式員不需要知道或在意該序列的底層究竟如何實作。此外迭代器是所謂的「輕量級」物件：產生的代價極小。基於這個理由，你常會發現它有一些看似奇怪的限制，例如某些迭代器只能單向移動。

Java **Iterator** 便是這樣一個例子，它也具有某種限制。你沒有辦法對它做太多事情，不過你可以：

1. 呼叫 **iterator()**，要求容器交給你一個 **Iterator**。當你第一次呼叫 **Iterator** 的 **next()** 時，它將會回傳序列中的第一個元素。

2. 呼叫 **next()** 取得序列中的下一個元素。

3. 呼叫 **hasNext()** 檢查序列中是否還有其他元素。

4. 呼叫 **remove()** 移去迭代器最新（最近）傳回的元素。

就這樣。它是一份簡單的實作品，但極具威力（對 **Lists** 來說，還有一個更複雜的 **ListIterator**）。爲了觀察其運作方式，讓我們重新探討本章先前的 **CatsAndDogs.java** 程式。在原先版本中，**get()** 被用來選擇序列內的元素，以下新版則採用了 **Iterator**：

```
//: c09:CatsAndDogs2.java
// Simple container with Iterator.
import java.util.*;

public class CatsAndDogs2 {
  public static void main(String[] args) {
    ArrayList cats = new ArrayList();
    for(int i = 0; i < 7; i++)
      cats.add(new Cat(i));
    Iterator e = cats.iterator();
    while(e.hasNext())
      ((Cat)e.next()).print();
  }
} ///:~
```

你會發現，最後幾行如今改以 **Iterator** 走訪序列內容，而不再使用 **for** 迴圈。有了 **Iterator**，你不再需要關心容器內實際有多少個元素，是的，**hasNext()** 和 **next()** 會爲你處理這個問題。

下面是另一個例子，假設我們要撰寫一個通用性的列印函式：

```
//: c09:HamsterMaze.java
// Using an Iterator.
import java.util.*;

class Hamster {
  private int hamsterNumber;
  Hamster(int i) { hamsterNumber = i; }
  public String toString() {
    return "This is Hamster #" + hamsterNumber;
  }
}
```

```java
class Printer {
  static void printAll(Iterator e) {
    while(e.hasNext())
      System.out.println(e.next());
  }
}

public class HamsterMaze {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 3; i++)
      v.add(new Hamster(i));
    Printer.printAll(v.iterator());
  }
} ///:~
```

仔細觀察 **printAll()**，請注意，其中並沒有任何和序列類型有關的資訊。你的手上只有 **Iterator**，面對序列你也只要知道這個 **Iterator** 就好了：你可以透過它取得下一個物件，你可以知道是否已經到達序列尾端。以這種方式對待容器，「在其上進行走訪，藉以在每個元素身上執行某個動作」的想法，相當具有威力，而且在本書中到處可見。

上述這個例子甚至更爲一般化，因爲它會暗自使用 **Object.toString()**。是的，**println()** 被重載以適用 **Object** 和任何基本型別，而在每一種情形下，**String** 都會被自動產生 — 由於呼叫了合適的 **toString()** 之故。

雖然沒有必要，但你也可以手動完成轉型，效果和呼叫 **toString()** 相同：

```java
System.out.println((String)e.next());
```

一般而言，你會想要進行一些動作，而不只是呼叫 **Object** 函式而已。所以你還是會碰到型別轉換的問題。你得假設你已經取得某個特定型別的序列的 **Iterator**，並將你所獲得的物件轉型至該型別。如果轉型錯誤，會收到執行期異常。

非預期的遞迴（Unintended recursion）

由於 Java 標準容器都繼承自 **Object**（和其他所有 classes 一樣），所以它們也具備了 **toString()**。這個函式可被覆寫以產生自定的 **String** 表示式。在這個 **String** 表示式中，可含括其所持有的物件。例如在 **ArrayList** 中，其 **toString()** 會走訪 **ArrayList** 的每一個元素並呼叫其 **toString()**。假設你想要印出你的 class 的位址，只要參考 **this** 就好了（C++程式員更會傾向於這個方法）：

```
//: c09:InfiniteRecursion.java
// Accidental recursion.
import java.util.*;

public class InfiniteRecursion {
  public String toString() {
    return " InfiniteRecursion address: "
      + this + "\n";
  }
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++)
      v.add(new InfiniteRecursion());
    System.out.println(v);
  }
} ///:~
```

但如果你只是產生一個 **InfiniteRecursion** 物件，然後列印它，你會收到無止盡的一連串異常回報。如果你將 **InfiniteRecursion** 物件置於 **ArrayList** 中，並印出該 **ArrayList**，也會發生同樣的問題。問題出在哪裡？在於 **String**s 的自動型別轉換。當你寫下這樣的程式：

```
"InfiniteRecursion address: " + this
```

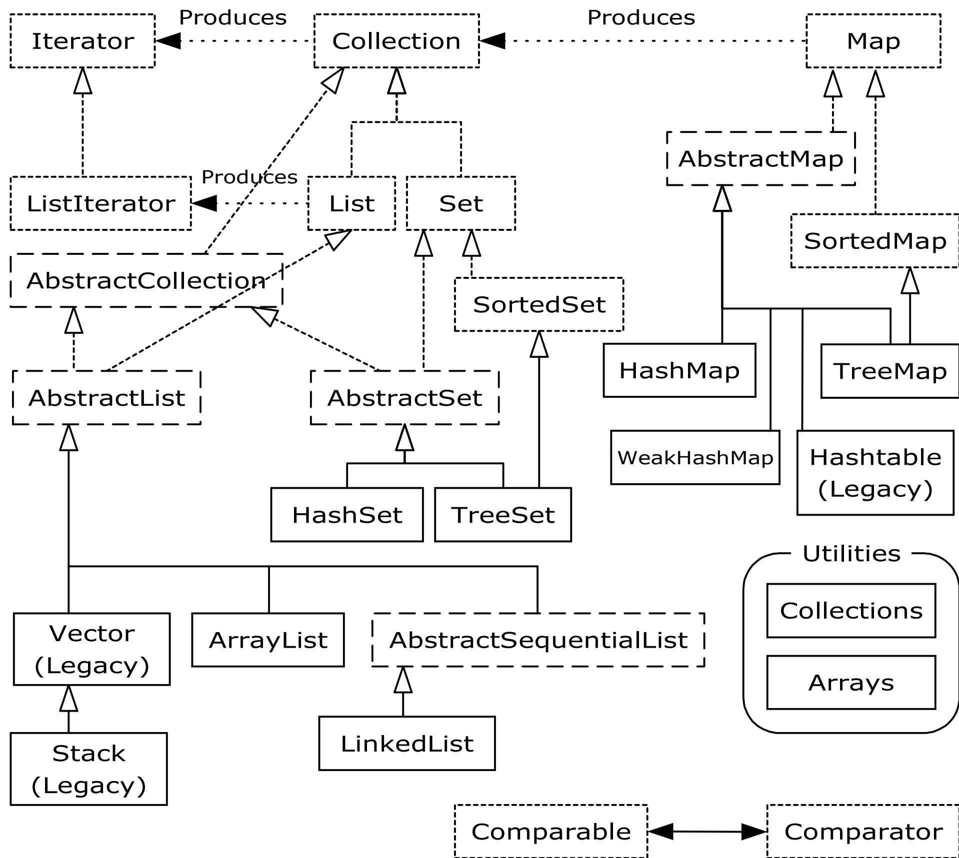
編譯器看到 **String** 之後緊跟著一個 '+' 號，而後緊接著一個 non-**String**，編譯器便試著將 **this** 轉換為 **String**。它會呼叫 **toString()** 來進行轉換，而這個動作會引發遞迴呼叫。

如果你想在這個例子中印出物件的位址，解決方法就是呼叫 **Object** 的 **toString()**。是的，就是如此。

所以，請不要使用 **this**，請改用 **super.toString()**。注意，只有當你直接繼承 **Object** 或你的 parent classes 都不曾覆寫 **toString()** 時，這個作法才管用。

容器分類學（Container taxonomy）

視程式編寫時的需要，**Collectiopns** 和 **Maps** 可被各種不同的方式實作出來。以下這張 Java 2 容器分類圖對於你的認知應該很有幫助：



乍見之下此圖可能令人不知所措。但你必然還是會發現到，實際上只有三種容器組件：**Map**、**List**、**Set**。而且每一種只有 2~3 個實作物。一旦你了解這一點，這些容器就不會那麼令人望而怯步了。

短虛線方塊代表 **interface**s，長虛線方塊代表 **abstract** classes，實線方塊則是一般（具象）的 classes。虛線箭頭表示「實作出某個 **interface**」的 class，或代表「僅部份實作出某 **interface**」的某個 **abstract** class。實線箭頭表示某個 class 可產生「箭頭所指的那個 class」的物件，例如所有 **Collection** 都可以產生 **Iterator**，而 **List** 可以產生 **ListIterator**（也可以產生一般的 **Iterator**，因為 **List** 繼承自 **Collection**）。

和「物件持有」相關的 interfaces 包括 **Collection**、**List**、**Set**、**Map**。理想情況下你所撰寫的程式大多只會和這些 interfaces 溝通，而且只有在產生它們的時候才會指定確切類型。所以你可以透過下列方式產生 **List**：

```
List x = new LinkedList();
```

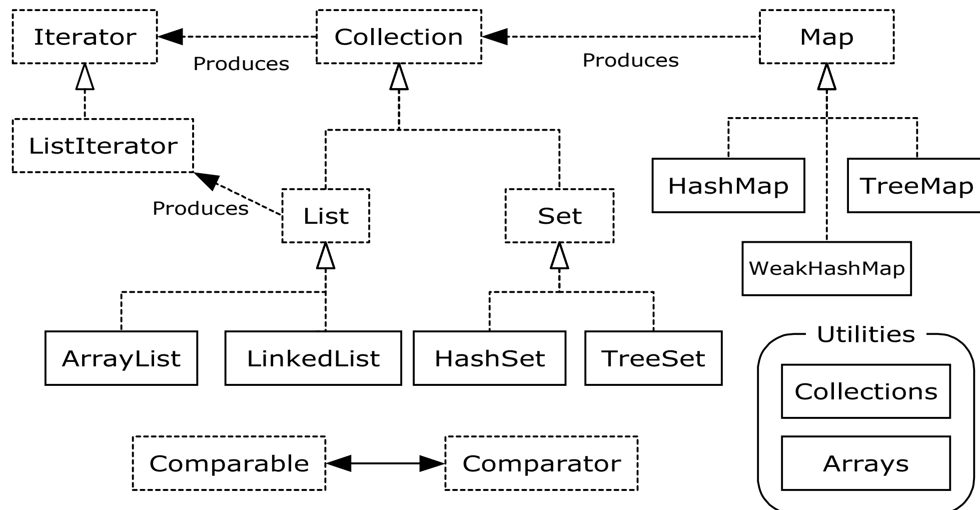
當然，你也可以令 **x** 為一個 **LinkedList**（而不是一個泛化的 **List**），並隱藏 **x** 的確切型別資訊。使用 **interface** 的美好之處在於，如果你想要改用另一種實作物，只要在產生物件的地點加以改變即可，像這樣：

```
List x = new ArrayList();
```

程式碼的其餘部份仍然可以維持不變（這種泛型性質的某一部份可由迭代器達成目的）。

在類別階層架構中，你會發現有些 classes 的名稱以 "**Abstract**" 為首。乍看之下它們令人困惑。事實上它們只是一些簡單工具，局部實作出某個 interface 罷了。例如，假設你想要製作自己的 **Set**，你可能不會從 **Set** 下手實作出所有函式，你可能會繼承 **AbstractSet**，因而得以花費最小力氣來製作新 class。不過 Java 容器庫提供的功能基本上可以滿足你的所有需求。所以，以我所設定的目標而言，你可以省略所有以 "**Abstract**" 為首的 classes。

因此，當你觀察這張分類圖時，只要關心最頂端的 **interface**s，以及所有 concrete classes（實線方塊）即可。通常你會產生那些 concrete class 物件，並將它向上轉型至相應的 **interface**，然後在程式碼的其他地方使用那個 **interface**。此外，撰寫新碼時你無需考慮舊有的元素。因此容器分類圖可被大幅簡化為：



請注意，這張分類圖只含括一般用途會用到的 interfaces 和 classes，以及本章關注的元素。

下面便是一個實例，將 **String** 物件填入 **Collection**（此處爲 **ArrayList**）之中，然後印出每個元素：

```
//: c09:SimpleCollection.java
// A simple example using Java 2 Collections.
import java.util.*;

public class SimpleCollection {
  public static void main(String[] args) {
    // Upcast because we just want to
    // work with Collection features
    Collection c = new ArrayList();
    for(int i = 0; i < 10; i++)
      c.add(Integer.toString(i));
    Iterator it = c.iterator();
    while(it.hasNext())
      System.out.println(it.next());
  }
} ///:~
```

main() 第一行產生一個 **ArrayList** 物件，並向上轉型至 **Collection**。由於這個例子只使用 **Collection** 的成員，所以繼承自 **Collection** 的任何 class 的物件都可有效運作。不過最常被用到的一種 **Collection** 還是 **ArrayList**。

add() 一如其名，會將新元素置入 **Collection**。然而說明文件中很謹慎地說了：**add()**「保證容器一定含入被指定的元素」。這其實說的是 **Set**，因爲 **Set** 只有在元素不重複的情況下才會加入新元素。使用 **ArrayList** 或其他類型的 **List** 時，**add()** 一律意味「將元素置入」，因爲 **Lists** 並不在乎元素是否重複。

所有 **Collections** 都會透過其 **iterator()** 產生一個 **Iterator**。上述例子會產生一個 **Iterator** 並以之走訪 **Collection**，印出每一個元素。

Collection 的機能

下表顯示你能夠對 **Collection** 做的所有動作（不包括因繼承 **Object** 而自動具備的函式）。因此 **Set** 和 **List** 也都具備了同樣的功能（**List** 還具備其他功能）。**Map** 並不繼承自 **Collection**，所以應該被個別看待。

boolean add(Object)	確保容器將持有「引數所代表的物件」。如果它沒能將引數加入，就回傳 **false**。（這是個可有可無的函式，本章稍後解釋）
boolean addAll(Collection)	將引數中的所有元素都加入容器內。只要加入了任一元素，就回傳 **true**（也是可有可無的，"optional"）
void clear( )	移除（remove）容器內的所有元素。（可有可無的，"optional"）
boolean contains(Object)	如果容器內含引數所代表的物件，就回傳 **true**。
boolean containsAll(Collection)	如果容器內含引數所含的所有元素，就回傳 **true**。

boolean isEmpty( )	如果容器內未含任何元素，回傳 **true**。
Iterator iterator( )	回傳一個 **Iterator**，你可以使用這個 **Iterator** 來走訪容器。
boolean remove(Object)	如果引數值位於容器內，便移除該元素（的其中之一）。如果確實發生移除動作，回傳 **true**。（可有可無的，"optional"）
boolean removeAll(Collection)	移除引數容器中的所有元素。如果發生移除動作，就回傳 **true**。（可有可無的，"optional"）
boolean retainAll(Collection)	只保留引數容器內的元素 — 採用集合理論中的 intersection（交集）。如果確實發生更動，回傳 **true**。（可有可無的，"optional"）
int size( )	回傳容器中的元素個數。
Object[] toArray( )	回傳一個 array，內含容器的所有元素。
Object[] toArray(Object[] a)	回傳一個 array，內含容器的所有元素。array 的元素型別和引數 **a** 的元素型別相同，非單純之 **Object**（你得自行將 array 轉爲正確型別）。

請注意，這裡並未提供隨機存取所需的 **get()** 函式，因爲 **Collection** 涵蓋 **Set**，而 **Set** 會維護它自己的內部排列（這就使得隨機存取不具意義）。因此如果你想檢視 **Collection** 內的所有元素，得使用迭代器才行；這也是唯一可用來取回元素的方法。

以下示範上述所有函式。再次我要說，這些函式都可作用於繼承自 **Collection** 的所有 classes 身上，而 **ArrayList** 則被用來做爲一種「最小公約數」：

```
//: c09:Collection1.java
// Things you can do with all Collections.
```

```java
import java.util.*;
import com.bruceeckel.util.*;

public class Collection1 {
  public static void main(String[] args) {
    Collection c = new ArrayList();
    Collections2.fill(c,
      Collections2.countries, 10);
    c.add("ten");
    c.add("eleven");
    System.out.println(c);
    // Make an array from the List:
    Object[] array = c.toArray();
    // Make a String array from the List:
    String[] str =
      (String[])c.toArray(new String[1]);
    // Find max and min elements; this means
    // different things depending on the way
    // the Comparable interface is implemented:
    System.out.println("Collections.max(c) = " +
      Collections.max(c));
    System.out.println("Collections.min(c) = " +
      Collections.min(c));
    // Add a Collection to another Collection
    Collection c2 = new ArrayList();
    Collections2.fill(c2,
      Collections2.countries, 10);
    c.addAll(c2);
    System.out.println(c);
    c.remove(CountryCapitals.pairs[0][0]);
    System.out.println(c);
    c.remove(CountryCapitals.pairs[1][0]);
    System.out.println(c);
    // Remove all components that are in the
    // argument collection:
    c.removeAll(c2);
    System.out.println(c);
    c.addAll(c2);
    System.out.println(c);
    // Is an element in this Collection?
    String val = CountryCapitals.pairs[3][0];
```

```
    System.out.println(
      "c.contains(" + val  + ") = "
      + c.contains(val));
    // Is a Collection in this Collection?
    System.out.println(
      "c.containsAll(c2) = "+ c.containsAll(c2));
    Collection c3 = ((List)c).subList(3, 5);
    // Keep all the elements that are in both
    // c2 and c3 (an intersection of sets):
    c2.retainAll(c3);
    System.out.println(c);
    // Throw away all the elements
    // in c2 that also appear in c3:
    c2.removeAll(c3);
    System.out.println("c.isEmpty() = " +
      c.isEmpty());
    c = new ArrayList();
    Collections2.fill(c,
      Collections2.countries, 10);
    System.out.println(c);
    c.clear(); // Remove all elements
    System.out.println("after c.clear():");
    System.out.println(c);
  }
} ///:~
```

這個程式產生一些 **ArrayList**s 來儲存不同組的資料，並將它向上轉型至 **Collection** 物件。所以很明顯，除了 **Collection** interface 之外不會用到其他東西。**main()** 只是以簡單的練習來測試 **Collection** 中的所有函式。

以下數節說明 **List**、**Set**、**Map** 的各式各樣實作，並指出哪一種是你的預設選擇（表格中帶有星號者）。請注意，老舊的（Java2 以前的）classes 如 **Vector**、**Stack**、**Hashtable** 都未在此處討論，因爲任何情況下你最好還是使用 Java 2 容器。

List 的機能

基本的 **List** 極易使用，就和你目前所見的 **ArrayList** 使用方式一樣。雖然大多數時候你大概只會以 **add()** 置入元素，以 **get()** 一次取出一個元素、以 **iterator()** 獲得一個指向序列的 **Iterator**，但還有一組函式可能派上用場。

此外，實際上有兩類 **List**：功能一般的 **ArrayList**，優點在於可隨機存取其中元素；功能較強的 **LinkedList**（並非爲了快速隨機存取而設計，具備一組更通用的函式）。

List (interface)	次序（order）是 **List** 最重要的特性；它保證以某種特定次序來維護元素。**List** 爲 **Collection** 加入了一些函式，使它得以在 **List** 內進行安插和移除動作（建議你只在 **LinkedList** 上這麼做）。**List** 會產生 **ListIterator**，透過它你可以從兩個方向來對 **List** 進行走訪，也可以在 **List** 之內進行元素的安插和移除。
ArrayList*	以 array 實作完成的 **List**。允許快速隨機存取，但是當元素的安插或移除發生於 **List** 中央位置時，效率便很差。面對 **ArrayList**，你應該只拿 **ListIterator** 來進行向後或向前走訪動作，而不應該拿它來進行元素安插和移除動作，因爲後者所花的代價遠較 **LinkedList** 高昂。
LinkedList	提供最佳循序存取，以及成本低廉的「**List** 中央位置元素安插和移除」。隨機存取動作則相對緩慢（如果想進行隨機存取，請改用 **ArrayList**）。它還具備 **addFirst()**、**addLast()**、**getFirst()**、**getLast()**、**removeFirst()**、**removeLast()**（這些函式並未定義於任何一個 interface 或 base classes 中），使其可被拿來當做 stack、queue、或 deque。

下例中的每個函式都涵蓋了不同類型的動作，包括：所有 **List** 都可執行的動作（**basicTest()**）、透過 **Iterator** 走訪（**iterMotion()**）、透過

Iterator 更動（**iterManipulation()**）、觀察 **List** 更動後的結果（**testVisual()**）、以及僅可用於 **LinkedLists** 的一些操作。

```
//: c09:List1.java
// Things you can do with Lists.
import java.util.*;
import com.bruceeckel.util.*;

public class List1 {
  public static List fill(List a) {
    Collections2.countries.reset();
    Collections2.fill(a,
      Collections2.countries, 10);
    return a;
  }
  static boolean b;
  static Object o;
  static int i;
  static Iterator it;
  static ListIterator lit;
  public static void basicTest(List a) {
    a.add(1, "x"); // Add at location 1
    a.add("x"); // Add at end
    // Add a collection:
    a.addAll(fill(new ArrayList()));
    // Add a collection starting at location 3:
    a.addAll(3, fill(new ArrayList()));
    b = a.contains("1"); // Is it in there?
    // Is the entire collection in there?
    b = a.containsAll(fill(new ArrayList()));
    // Lists allow random access, which is cheap
    // for ArrayList, expensive for LinkedList:
    o = a.get(1); // Get object at location 1
    i = a.indexOf("1"); // Tell index of object
    b = a.isEmpty(); // Any elements inside?
    it = a.iterator(); // Ordinary Iterator
    lit = a.listIterator(); // ListIterator
    lit = a.listIterator(3); // Start at loc 3
    i = a.lastIndexOf("1"); // Last match
    a.remove(1); // Remove location 1
```

```
    a.remove("3"); // Remove this object
    a.set(1, "y"); // Set location 1 to "y"
    // Keep everything that's in the argument
    // (the intersection of the two sets):
    a.retainAll(fill(new ArrayList()));
    // Remove everything that's in the argument:
    a.removeAll(fill(new ArrayList()));
    i = a.size(); // How big is it?
    a.clear(); // Remove all elements
  }
  public static void iterMotion(List a) {
    ListIterator it = a.listIterator();
    b = it.hasNext();
    b = it.hasPrevious();
    o = it.next();
    i = it.nextIndex();
    o = it.previous();
    i = it.previousIndex();
  }
  public static void iterManipulation(List a) {
    ListIterator it = a.listIterator();
    it.add("47");
    // Must move to an element after add():
    it.next();
    // Remove the element that was just produced:
    it.remove();
    // Must move to an element after remove():
    it.next();
    // Change the element that was just produced:
    it.set("47");
  }
  public static void testVisual(List a) {
    System.out.println(a);
    List b = new ArrayList();
    fill(b);
    System.out.print("b = ");
    System.out.println(b);
    a.addAll(b);
    a.addAll(fill(new ArrayList()));
    System.out.println(a);
    // Insert, remove, and replace elements
```

```
    // using a ListIterator:
    ListIterator x = a.listIterator(a.size()/2);
    x.add("one");
    System.out.println(a);
    System.out.println(x.next());
    x.remove();
    System.out.println(x.next());
    x.set("47");
    System.out.println(a);
    // Traverse the list backwards:
    x = a.listIterator(a.size());
    while(x.hasPrevious())
      System.out.print(x.previous() + " ");
    System.out.println();
    System.out.println("testVisual finished");
  }
  // There are some things that only
  // LinkedLists can do:
  public static void testLinkedList() {
    LinkedList ll = new LinkedList();
    fill(ll);
    System.out.println(ll);
    // Treat it like a stack, pushing:
    ll.addFirst("one");
    ll.addFirst("two");
    System.out.println(ll);
    // Like "peeking" at the top of a stack:
    System.out.println(ll.getFirst());
    // Like popping a stack:
    System.out.println(ll.removeFirst());
    System.out.println(ll.removeFirst());
    // Treat it like a queue, pulling elements
    // off the tail end:
    System.out.println(ll.removeLast());
    // With the above operations, it's a dequeue!
    System.out.println(ll);
  }
  public static void main(String[] args) {
    // Make and fill a new list each time:
    basicTest(fill(new LinkedList()));
    basicTest(fill(new ArrayList()));
```

```
    iterMotion(fill(new LinkedList()));
    iterMotion(fill(new ArrayList()));
    iterManipulation(fill(new LinkedList()));
    iterManipulation(fill(new ArrayList()));
    testVisual(fill(new LinkedList()));
    testLinkedList();
  }
} ///:~
```

在 **basicTest()** 和 **iterMotion()** 中，所有呼叫動作只是用來示範正確的語法；收到回傳值後並不使用之。某些情況下甚至不接收回傳值，因爲通常不用它。使用這些函式前，你應該檢閱 *java.sun.com* 的線上文件，仔細看看其使用方式。

根據 LinkedList 製作一個 stack

stack（堆疊）有時候被稱爲一種「後進先出（*last-in, first-out*，LIFO）」容器。也就是說，被你最後推入（push）stack 內元素，便是你從 stack 最先取出（pop）的元素。和 Java 的所有其他容器一樣，你所推入和取出的通通都是 **Object**s，所以你必須對取出的物件進行轉型，除非你只想使用 **Object** 的功能。

LinkedList 擁有一些函式，可直接實作出 stack，所以你可以直接運用 **LinkedList** 而無需重新設計一個 stack class。不過這樣一個 stack class 有時候能夠把事情說得更清楚一些，所以還是有其價值：

```
//: c09:StackL.java
// Making a stack from a LinkedList.
import java.util.*;
import com.bruceeckel.util.*;

public class StackL {
  private LinkedList list = new LinkedList();
  public void push(Object v) {
    list.addFirst(v);
  }
  public Object top() { return list.getFirst(); }
  public Object pop()
    return list.removeFirst();
  }
```

```
  public static void main(String[] args) {
    StackL stack = new StackL();
    for(int i = 0; i < 10; i++)
      stack.push(Collections2.countries.next());
    System.out.println(stack.top());
    System.out.println(stack.top());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
  }
} ///:~
```

如果你只是需要 stack 的行爲，此處不需繼承，因爲繼承會使產生出來的 class 具有「**LinkedList** 的其餘所有函式」。稍後你會看到，這正是 Java 1.0 程式庫設計者在 **Stack** 上所犯的嚴重錯誤。

根據 LinkedList 製作一個 queue

queue 是一種「先進先出（*first-in, first-out*，FIFO）」容器。也就是說，你將物件於某端置入，於另一端取出。因此元素的置入次序恰恰就是其被取出的次序。**LinkedList** 擁有一些函式可直接支援 queue，你可以把它們應用於 **Queue** class 之中：

```
//: c09:Queue.java
// Making a queue from a LinkedList.
import java.util.*;

public class Queue {
  private LinkedList list = new LinkedList();
  public void put(Object v) { list.addFirst(v); }
  public Object get()
    return list.removeLast();
  }
  public boolean isEmpty()
    return list.isEmpty();
  }
  public static void main(String[] args) {
    Queue queue = new Queue();
    for(int i = 0; i < 10; i++)
      queue.put(Integer.toString(i));
```

```
    while(!queue.isEmpty())
      System.out.println(queue.get());
  }
} ///:~
```

你也可以利用 **LinkedList** 輕易製作出一個 deque（雙邊開口的 queue）。它就像一個 queue，只不過兩端都可置入和移除元素。

Set 的機能

Set 擁有和 **Collection** 一模一樣的 interfaces，所以它不像上述兩種 **List**s 一樣地有額外機能。**Set** 就是一個 **Collection**，只不過其行為不同罷了（這是繼承和多型的理想運用方式：展現不同行為）。**Set** 拒絕持有重複的元素實值（*value*）。如何制定所謂的元素「實值」，情況頗為複雜，稍後你便會看到。

Set (interface)	加至 **Set** 內的每個元素都必須獨一無二，不與其他元素重複；**Set** 不允許持有重複元素。每個元素都必須定義 **equals()** 以判斷所謂的獨一性。**Set** 具有和 **Collection** 一模一樣的 interface。**Set** interface 並不保證以任何特定次序來維護其元素。
HashSet*	一種把搜尋時間看得很重要的 **Set**s。所有元素都必須定義 **hashCode()**。
TreeSet	底層結構為 tree 的一種有序的（ordered）**Set**。這麼一來你便可以自 **Set** 中萃取出一個帶次序性的序列（ordered sequence）。

以下程式並不示範 **Set** 的所有機能，因為 **Set** interface 和 **Collection** 相同，已在先前範例中練習過了。這個範例所示範的是 **Set** 獨有的行為：

```
//: c09:Set1.java
// Things you can do with Sets.
import java.util.*;
import com.bruceeckel.util.*;
```

```
public class Set1 {
  static Collections2.StringGenerator gen =
    Collections2.countries;
  public static void testVisual(Set a) {
    Collections2.fill(a, gen.reset(), 10);
    Collections2.fill(a, gen.reset(), 10);
    Collections2.fill(a, gen.reset(), 10);
    System.out.println(a); // No duplicates!
    // Add another set to this one:
    a.addAll(a);
    a.add("one");
    a.add("one");
    a.add("one");
    System.out.println(a);
    // Look something up:
    System.out.println("a.contains(\"one\"): " +
      a.contains("one"));
  }
  public static void main(String[] args) {
    System.out.println("HashSet");
    testVisual(new HashSet());
    System.out.println("TreeSet");
    testVisual(new TreeSet());
  }
} ///:~
```

這個程式接受重複元素值，但是當你列印 **Set** 的內容，你會發現針對每一個值，**Set** 僅接受一份。

當你執行這個程式，你會注意到，**HashSet** 所維護的次序和 **TreeSet** 不相同。這是因爲兩者以不同的方式來儲存元素，使它們稍後還能找到該元素。**TreeSet** 會讓各元素保持排序（sorted）狀態，**HashSet** 則使用極適合快速搜尋的所謂 hashing 函式。開發自己的型別（types）時請務必記住，**Set** 需要某種方式以維護其元素次序，這意味你必須實作 **Comparable** interface，並定義 **compareTo()**。以下便是一例：

```
//: c09:Set2.java
// Putting your own type in a Set.
import java.util.*;
```

```
class MyType implements Comparable {
  private int i;
  public MyType(int n) { i = n; }
  public boolean equals(Object o) {
    return
      (o instanceof MyType)
      && (i == ((MyType)o).i);
  }
  public int hashCode() { return i; }
  public String toString() { return i + " "; }
  public int compareTo(Object o) {
    int i2 = ((MyType)o).i;
    return (i2 < i ? -1 : (i2 == i ? 0 : 1));
  }
}

public class Set2 {
  public static Set fill(Set a, int size) {
    for(int i = 0; i < size; i++)
      a.add(new MyType(i));
    return a;
  }
  public static void test(Set a) {
    fill(a, 10);
    fill(a, 10); // Try to add duplicates
    fill(a, 10);
    a.addAll(fill(new TreeSet(), 10));
    System.out.println(a);
  }
  public static void main(String[] args) {
    test(new HashSet());
    test(new TreeSet());
  }
} ///:~
```

本章稍後會介紹 **equals()** 和 **hashCode()** 的定義方式。使用 **HashSet** 和 **TreeSet** 時，你都得爲你的 class 定義 **equals()**，但只有當你的 class 將被置於 **HashSet** 時，才一定得定義 **hashCode()**（很有可能如此，因爲 **HashSet** 往往是你選擇 **Set** 實作對象時的第一選擇）。不過，爲了保

持良好的程式設計風格，當你覆寫 **equals()**，你應該一併覆寫 **hashCode()**。本章稍後會完整探討這個過程。

請注意我並未在 **compareTo()** 中使用簡單明瞭的型式：**return i-i2**。這種寫法是常見的錯誤，因爲只有當 **i** 和 **i2** 都是無正負號（**unsigned**，如果 Java 有關鍵字 **unsigned** 的話）的 **ints**，這種寫法才正確。面對 Java 之中帶正負號的 **int** 會出錯。原因是帶正負號的 **int**，其容量不足以表示兩個同型數值的相減結果。如果 **i** 是個夠大的正整數，**j** 是個夠大的負整數，那麼 **i-j** 的結果便會造成溢位，並回傳負值，這便導致了錯誤。

SortedSet

如果你有一個 **SortedSet**（**TreeSet** 是唯一一份實作），將保證其中元素處於排序狀態，並提供 **SortedSet** interface 內規劃的以下函式：

Comparator comparator()：產生一個「被這個 **Set** 所使用」的 **Comparator**。或是回傳 **null**，代表以自然方式排序。

Object first()：產生最低元素（lowest element）。

Object last()：產生最高元素（highest element）。

SortedSet subSet(fromElement, toElement)：產生 **Set** 子集，範圍從 **fromElement**（含）到 **toElement**（不含）。

SortedSet headSet(toElement)：產生 **Set** 子集，其中元素皆小於 **toElement**。

SortedSet tailSet(fromElement)：產生 **Set** 子集，其中元素皆大於或等於 **fromElement**。

Map 的機能

ArrayList 讓你得以使用數字來選擇一大群物件中的某一個。所以就某種意義來說，它將數字和物件產生關聯。如果你想要根據其他選擇條件，在一連串物件中進行挑選，又該如何？stack 便是一例：它的選擇條件是「最晚被推進 stack 的那一個」。把「在序列中挑選」的概念發揮到極致的便是所謂的 map，或稱爲 *dictionary*（字典）、*associative array*（關聯式

陣列）。觀念上它就是一個 **ArrayList**，但是並不以數字來搜尋元素，而是以另一個物件來進行搜尋！這在程式中往往是一種重要過程。

上述觀念在 Java 中以 **Map** interface 落實。**put(Object key, Object value)** 會將 *value* 加入，並將它關聯至 *key*（這就是你將用以做爲搜尋依據的東西）。**get(Object key)** 會傳回 *key* 所對應的 *value*。你也可以使用 **containsKey()** 和 **containsValue()** 來檢查 Map 內是否含有某個 *key* 或某個 *value*。

Java 標準程式庫有兩個不同類型的 **Maps**：**HashMap** 和 **TreeMap**。二者具有相同的 interface（因爲它們都實作了 **Map**），但效率表現卻不相同。如果你知道在執行 **get()** 時必須進行哪些動作，你就會發現，「在 **ArrayList** 中搜尋 *key*」這個動作的速度似乎不快。而這正是 **HashMap** 提高速度的地方。它不會緩慢地逐一搜尋某個 *key*，而會使用一種被稱爲「*hash code*（雜湊碼）」的特殊值。hash code 是一種「將物件內的某些資訊轉換爲幾乎獨一無二」的 **int**，用以代表那個物件。所有 Java 物件都可以產生 hash code，而且 **hashCode()** 是根源類別 **Object** 具備的函式。**HashMap** 會運用物件的 **hashCode()**，並利用它來快速找到 *key*。這種作法能夠帶來巨幅的效率提升[7]。

Map (interface)	維護 *key-value* 的關聯性，使你可以使用 *key* 來搜尋 *value*。
HashMap*	基於 hash table（雜湊表）完成的一個實作品。可用它來取代 **Hashtable**（譯註：Java 2 之前的一種容器）。可在常數時間內安插元素，或找出一組 *key-value* pair。透過其建構式，使用者可調整效能表現，因爲它允許你設定 *capacity*（容量）和 *load factor*（負載因子）

[7]如果這樣的加速結果仍然無法符合你對效能的要求，你便應該撰寫自己的 **Map**，並針對你的特定型別進行訂製，以避免因轉型而造成的延遲，並藉以加速表格搜尋。爲了達到更高效能，你可以參考 Donald Knuth 所著的《*The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition*》，以 array 來替換滿載的 bucket lists（譯註：這是 hash table 中實際用來儲存元素的一種結構，請參考資料結構專書中對於 hash table 的介紹）。運用 array 能夠獲得兩個好處：可針對磁碟空間的特性進行最佳化，並可省下「產生個別資料」和「進行垃圾回收」的大部份時間。

TreeMap	基於紅黑樹（red-black tree）完成的一個實作品。當你檢視其中的 *key* 或 *key-value* pairs 時，會以排序形式出現（前後次序由 **Comparable** 或 **Comparator** 決定，稍後介紹）。**TreeMap** 的特色便是讓你得以排序形式得到結果。**TreeMap** 是唯一具有 **subMap()** 的一個 **Map**，這個函式讓你得以回傳 tree 中的部份組成。

有時候，你也會需要知道 hashing 動作的執行細節，稍後我會加以探討。

以下範例用到了 **Collection2.fill()**，以及先前定義的測試資料集：

```
//: c09:Map1.java
// Things you can do with Maps.
import java.util.*;
import com.bruceeckel.util.*;

public class Map1 {
  static Collections2.StringPairGenerator geo =
    Collections2.geography;
  static Collections2.RandStringPairGenerator
    rsp = Collections2.rsp;
  // Producing a Set of the keys:
  public static void printKeys(Map m) {
    System.out.print("Size = " + m.size() +", ");
    System.out.print("Keys: ");
    System.out.println(m.keySet());
  }
  // Producing a Collection of the values:
  public static void printValues(Map m) {
    System.out.print("Values: ");
    System.out.println(m.values());
  }
  public static void test(Map m) {
    Collections2.fill(m, geo, 25);
    // Map has 'Set' behavior for keys:
    Collections2.fill(m, geo.reset(), 25);
    printKeys(m);
```

```
    printValues(m);
    System.out.println(m);
    String key = CountryCapitals.pairs[4][0];
    String value = CountryCapitals.pairs[4][1];
    System.out.println("m.containsKey(\"" + key +
      "\"): " + m.containsKey(key));
    System.out.println("m.get(\"" + key + "\"): "
      + m.get(key));
    System.out.println("m.containsValue(\""
      + value + "\"): " +
      m.containsValue(value));
    Map m2 = new TreeMap();
    Collections2.fill(m2, rsp, 25);
    m.putAll(m2);
    printKeys(m);
    key = m.keySet().iterator().next().toString();
    System.out.println("First key in map: "+key);
    m.remove(key);
    printKeys(m);
    m.clear();
    System.out.println("m.isEmpty(): "
      + m.isEmpty());
    Collections2.fill(m, geo.reset(), 25);
    // Operations on the Set change the Map:
    m.keySet().removeAll(m.keySet());
    System.out.println("m.isEmpty(): "
      + m.isEmpty());
  }
  public static void main(String[] args) {
    System.out.println("Testing HashMap");
    test(new HashMap());
    System.out.println("Testing TreeMap");
    test(new TreeMap());
  }
} ///:~
```

printKeys() 和 **printValues()** 不單只是有用的公用函式，它們同時也示範如何將 **Map** 的內容生成一個 **Collection**。**keySet()** 會將 **Map** 內的 *keys* 生成一個 **Set**。**values()** 的行爲類似，會產生一個 **Collection**，包含 **Map** 內的所有 *values*。請注意，*keys* 肯定是獨一無二的，但 *values* 卻

允許重複。由於這些 **Collections** 的背後支撐是 **Map**，所以對 **Collection** 所做的更動都會反映在相應的那個 **Map** 中。

程式的其餘部份示範了 **Map** 的每一種操作，並測試各種類型的 **Map**。

爲了示範 **HashMap** 的使用方式，請設想有個程式，它想要檢查 **Math.random()** 所產生的隨機亂數是否夠亂。理想情況下它應該產生一組完美的亂數分佈，爲了檢驗此事，你得產生許多亂數，並且計算落在不同區間內的亂數個數。**HashMap** 恰可解決這個問題，因爲它能夠將物件關聯至物件。本例之中，*value* 物件內含由 **Math.random()** 產生的數字，以及該數字的出現次數：

```
//: c09:Statistics.java
// Simple demonstration of HashMap.
import java.util.*;

class Counter
  int i = 1;
  public String toString()
    return Integer.toString(i);
  }
}

class Statistics {
  public static void main(String[] args) {
    HashMap hm = new HashMap();
    for(int i = 0; i < 10000; i++) {
      // Produce a number between 0 and 20:
      Integer r =
        new Integer((int)(Math.random() * 20));
      if(hm.containsKey(r))
        ((Counter)hm.get(r)).i++;
      else
        hm.put(r, new Counter());
    }
    System.out.println(hm);
  }
} ///:~
```

main() 之中每次產生一個亂數，就以 **Integer** 物件加以包裝，這才使得 object reference 可被用於 **HashMap** 中（是的，你無法將基本型別數值置入容器內，只有 object reference 才行）。**containsKey()** 會檢查 *key* 是否已在容器之內（也就是說此數字是否已經產生過了）。如果已在容器內，就使用 **get()** 取出該 *key* 相關聯的 *value* — 本例是個 **Counter** 物件。**Counter** 物件內的 **i** 值會被加一，代表該亂數值又被產生了一次。

如果未能找到 *key*，就呼叫 **put()**將新的 *key-value* pair 置於 **HashMap** 內。由於 **Counter** 在新誕生的時候會自動將變數 **i** 初始化爲 1，這便表示該亂數值首次出現。

只要將 **HashMap** 印出，便可顯示其中內容。**HashMap** 的 **toString()** 會走訪容器內的所有 *key-value* pairs，並呼叫它們的 **toString()**。**Integer.toString()** 已經預先定義好了，另外你也看到了 **Counter** 的 **toString()**。某次執行後的輸出結果如下（爲了版面考量，我加了一些斷行符號）：

```
{19=526, 18=533, 17=460, 16=513, 15=521, 14=495,
 13=512, 12=483, 11=488, 10=487, 9=514, 8=523,
 7=497, 6=487, 5=480, 4=489, 3=509, 2=503, 1=475,
 0=505}
```

你可能會猜想 class **Counter** 的必要性，看起來它似乎連外覆類別 **Integer** 的功能都不具備。爲什麼不使用 **int** 或 **Integer** 呢？呃，你不能使用 **int**，因爲所有容器都只能持有 **Object** reference。對你來說，了解容器的特性之後，外覆類別才會稍具意義，因爲你無法將基本型別數值置入容器內。不過，使用 Java 的外覆類別時，你只能在初始化時設定其值，而後讀取其值。換句話說你沒有辦法在產生外覆類別的物件之後改變其值。這個特性使 **Integer** 外覆類別立即被判出局，所以我們得撰寫新的 class 來滿足我們的需求。

SortedMap

如果你採用 **SortedMap**（**TreeMap** 是唯一可用的一份實作），保證會對元素的鍵值（keys）進行排序，因而允許 **SortedMap** 提供一些額外機能：

Comparator comparator()：產生一個被此 **Map** 所使用的 **Comparator**。或回傳 **null**，表示將以自然方式進行排序。

Object firstKey()：產生最低的（lowest）*key*。

Object lastKey()：產生最高的（highest）*key*。

SortedMap subMap(fromKey, toKey): 產生此 **Map** 的一個子集，範圍從 **fromKey**（含）到 **toKey**（不含）。

SortedMap headMap(toKey)：產生此 **Map** 的一個子集，其內各元素的 *key* 皆小於 **toKey**。

SortedMap tailMap(fromKey)：產生此 **Map** 的一個子集，其內各元素的 *key* 皆大於或等於 **fromKey**。

Hashing 和 hash codes

上個例子中，標準程式庫提供的一個 class（**Integer**）被用來做爲 **HashMap** 的 *key*。這很適合，因爲它具備所有必要性質，使它做爲 *key* 時可以運作正確。但是當你以自己撰寫的 class 做爲 *key* 時，會碰上一個 **HashMaps** 的常見問題。舉個例子，假設有個氣象預測系統，以 **Groundhog**（土撥鼠）物件對 **Prediction**（預報）進行比對。解決方法似乎很簡單：撰寫兩個 classes 並使用 **Groundhog** 做爲 *key*，以 **Prediction** 做爲 *value*：

```
//: c09:SpringDetector.java
// Looks plausible, but doesn't work.
import java.util.*;

class Groundhog {
  int ghNumber;
  Groundhog(int n) { ghNumber = n; }
}
```

```
class Prediction {
  boolean shadow = Math.random() > 0.5;
  public String toString() {
    if(shadow)
      return "Six more weeks of Winter!";
    else
      return "Early Spring!";
  }
}

public class SpringDetector {
  public static void main(String[] args) {
    HashMap hm = new HashMap();
    for(int i = 0; i < 10; i++)
      hm.put(new Groundhog(i), new Prediction());
    System.out.println("hm = " + hm + "\n");
    System.out.println(
      "Looking up prediction for Groundhog #3:");
    Groundhog gh = new Groundhog(3);
    if(hm.containsKey(gh))
      System.out.println((Prediction)hm.get(gh));
    else
      System.out.println("Key not found: " + gh);
  }
} ///:~
```

每個 **Groundhog** 都被賦予一個編號，所以如果你想要在 **HashMap** 中搜尋 **Prediction**，只要說「給我那個和 **Groundhog** #3 相關的 **Prediction**」。**Prediction** class 內含一個 **boolean** 値和一個 **toString()**，前者利用 **Math.random()** 完成初始化，後者爲你解釋結果。**main()** 將 **Groundhog**s 和其相應的 **Prediction**s 置入 **HashMap** 內。程式會將 **HashMap** 的內容印出，所以你可以觀察它是否的確被置入一些內容。然後，編號 3 那個 **Groundhog** 被用來做爲 *key*，用以搜尋對應的預報內容（你可以看到，它一定在 **Map** 之中）。

看起來似乎很簡單，結果卻不正確。問題出在 **Groundhog** 乃繼承自共通的根類別 **Object**（如果你未指定 base class，任何 classes 都會自動繼承自 **Object**。因此所有 class 最終都繼承自 **Object**）。**Object** 的

hashCode() 會被用來產生每個物件的 hash code，而且在預設情況下它直接使用其物件的記憶體位址。因此第一個 **Groundhog(3)** 所產生的 hash code 並不會和我們用來搜尋的第二個 **Groundhog(3)** 的 hash code 相等。

你可能會認爲，只要以適當的 **hashCode()** 加以覆寫（override）就好了。但只是這麼做還不行，除非你也覆寫同屬於 **Object** 的 **equals()**。當 **HashMap** 試著判斷你所給定的 *key* 是否等於表格中的某個 *key* 時，便會透過這個函式來判斷。同樣地，預設的 **Object.equals()** 只會比較物件的記憶體位址，所以兩個 **Groundhog(3)** 並不相同。

因此，如果你在 **HashMap** 中使用你自己撰寫的 classes 做爲 *key*，你一定得同時覆寫 **hashCode()** 和 **equals()**，像這樣：

```
//: c09:SpringDetector2.java
// A class that's used as a key in a HashMap
// must override hashCode() and equals().
import java.util.*;

class Groundhog2 {
  int ghNumber;
  Groundhog2(int n) { ghNumber = n; }
  public int hashCode() { return ghNumber; }
  public boolean equals(Object o) {
    return (o instanceof Groundhog2)
      && (ghNumber == ((Groundhog2)o).ghNumber);
  }
}

public class SpringDetector2 {
  public static void main(String[] args) {
    HashMap hm = new HashMap();
    for(int i = 0; i < 10; i++)
      hm.put(new Groundhog2(i),new Prediction());
    System.out.println("hm = " + hm + "\n");
    System.out.println(
      "Looking up prediction for groundhog #3:");
    Groundhog2 gh = new Groundhog2(3);
    if(hm.containsKey(gh))
```

```
        System.out.println((Prediction)hm.get(gh));
    }
} ///:~
```

請注意，這個程式使用了前例的 **Prediction** class，所以你得先行編譯 **SpringDetector.java**，否則在你試著編譯 **SpringDetector2.java** 時會收到編譯錯誤訊息。

Groundhog2.hashCode() 會回傳其編號以資識別。本例之中程式員必須保證，沒有任何兩個 **Groundhog** 物件具有相同的編號。**hashCode()** 不見得一定要回傳這獨一無二的編號（本章稍後你會更明白），但是 **equals()** 一定要能夠精準判斷兩個物件是否相等。

即使 **equals()** 只是檢查引數是否爲 **Groundhog2** 實體（以關鍵字 **instanceof** 進行檢查，詳見第 12 章），**instanceof** 實際上還會暗中進行第二個檢查，也就是檢查該物件是否爲 **null**，因爲 **instanceof** 會在其左方引數爲 **null** 時回傳 **false**。假設此物件的型別正確而且不是 **null**，那麼便會依據實際的 **ghNumbers** 進行比較。這時候，執行這個程式便可以看到正確的輸出結果。

當你撰寫用於 **HashSet** 的 class 時，你同樣得留意發生於 **HashMap** 身上的相同問題。

覆寫 hashCode()

上例只是正確解決問題的第一步。它說明了如果你不覆寫你所使用的 *key* 的 **hashCode()** 和 **equals()**，那麼任何在其底層運用 hashed 機制的資料結構（**HashSet** 或 **HashMap**）都將無法正確處理你所提供的 *key*。如果想妥善解決這個問題，你得了解 hashed 資料結構的運作方式才行。

首先，讓我們思考 hashing 背後的動機：透過某個物件搜尋另一個物件。但是你也可以使用 **TreeSet** 或 **TreeMap** 來達到同樣的目的呀。你甚至可以實作自己的 **Map**。爲了達到這個目的，必須提供 **Map.entrySet()** 來產生一組 **Map.Entry** 物件。**MPair** 會被定義爲新的 **Map.Entry** 型別。

由於我們想將它置於 **TreeSet** 內，所以它得實作 **equals()**，而且得是 **Comparable** 才行：

```
//: c09:MPair.java
// A Map implemented with ArrayLists.
import java.util.*;

public class MPair
implements Map.Entry, Comparable {
  Object key, value;
  MPair(Object k, Object v) {
    key = k;
    value = v;
  }
  public Object getKey() { return key; }
  public Object getValue() { return value; }
  public Object setValue(Object v){
    Object result = value;
    value = v;
    return result;
  }
  public boolean equals(Object o) {
    return key.equals(((MPair)o).key);
  }
  public int compareTo(Object rv) {
    return ((Comparable)key).compareTo(
      ((MPair)rv).key);
  }
} ///:~
```

注意，比較的對象只是 *keys*，所以重複的 *values* 完全可以被接受。

以下例子使用一對 **ArrayLists** 實作出一個 **Map**：

```
//: c09:SlowMap.java
// A Map implemented with ArrayLists.
import java.util.*;
import com.bruceeckel.util.*;

public class SlowMap extends AbstractMap {
  private ArrayList
```

```
    keys = new ArrayList(),
    values = new ArrayList();
  public Object put(Object key, Object value) {
    Object result = get(key);
    if(!keys.contains(key)) {
      keys.add(key);
      values.add(value);
    } else
      values.set(keys.indexOf(key), value);
    return result;
  }
  public Object get(Object key) {
    if(!keys.contains(key))
      return null;
    return values.get(keys.indexOf(key));
  }
  public Set entrySet() {
    Set entries = new HashSet();
    Iterator
      ki = keys.iterator(),
      vi = values.iterator();
    while(ki.hasNext())
      entries.add(new MPair(ki.next(), vi.next()));
    return entries;
  }
  public static void main(String[] args) {
    SlowMap m = new SlowMap();
    Collections2.fill(m,
      Collections2.geography, 25);
    System.out.println(m);
  }
} ///:~
```

put() 很單純地只是將 *keys* 和 *values* 置於對應的 **ArrayLists** 中。**main()** 會載入 **SlowMap**，然後列印出來以顯示其運作。

這個例子告訴我們，撰寫新型的 **Map** 並不多麼難。但是一如其名稱所示，**SlowMap** 執行的並不快，所以如果你有其他替代方案，大概不會用它。問題出在 *key* 的搜尋上：由於不具任何順序，所以只得採用最簡單的線性搜尋，而這種搜尋方式正是最慢的一種。

hashing 的目的完全是爲了速度：hashing 讓搜尋動作得以快速進行。由於 *key* 的搜尋是速度瓶頸所在，如果我們先將 *key* 排序，然後再使用 **Collecctions.binarySearch()** 執行搜尋，可能是這個問題的解法之一（本章最後有個練習，便是讓你完成這個解法）。

hashing 採取更進一步的解法：你只要將 *key* 儲存於「某處」，它便可以快速找到它。當你閱讀至本書此處，你知道，用來儲存一群元素的結構中，速度最快的當屬 array，所以 array 被用來表現 *key* 的資訊（請注意我說的是「*key* 的資訊」而非 *key* 本身）。同樣地，既然你已經進行到本書這一章，你應該已經明白，array 一經配置便無法改變容量大小，於是便引發了問題：我們想在 **Map** 中儲存任意數量的 *values*，但如果 *key* 的數目受限於 array 的容量，該當如何？

答案是，array 並不儲存 *key*。我們可以從 *key* 物件得出一個數字，用來做爲 array 的索引。這個數字即爲 **hashCode()**（計算機術語稱此爲 *hash function*）所產生的 *hash code*。**hashCode()** 定義於 **Object** 內，而且應該由你的 class 加以覆寫。爲了解決因 array 容量不變而引發的問題，多個 *keys* 可能會得出同一個索引值。也就是說可能會有「碰撞」（*collisions*）發生。由於這樣，array 的容量已無關緊要，因爲每個 *key* 物件都會被儲存於 array 的某處。

於是，搜尋某個 *value* 的過程，會從計算其 hash code 開始，然後使用此 hash code 做爲 array 的索引。如果你能夠保證不發生碰撞（當你只有固定個數的 *values* 時，這種情形便有可能），那就等於擁有一個「完美的 *hashing function*」，但這畢竟只是特例。其他時候，碰撞是透過所謂「外部鏈結」（*external chaining*）來解決：array 並不直接指向某個 *value*，而是指向一串 *values*。這些 *values* 以線性方式透過 **equals()** 進行搜尋。當然這種搜尋緩慢許多，但如果 hash function 夠好，同一個 slot 上只會有少數幾個 *values*（譯註：每個索引值代表（指向）一個 slot）。所以，無需搜尋整個 list，你就可以快速跳到正確的 slot，然後只要在少數幾個 *values* 之間進行比較，就可以找到搜尋目標。這種作法快多了，這也正是 **HashMap** 之所以快速的原因。

知道 hashing 的基本運作原則後，讓我們實作一個簡單的 hashed **Map**：

```
//: c09:SimpleHashMap.java
// A demonstration hashed Map.
import java.util.*;
import com.bruceeckel.util.*;

public class SimpleHashMap extends AbstractMap {
  // Choose a prime number for the hash table
  // size, to achieve a uniform distribution:
  private final static int SZ = 997;
  private LinkedList[] bucket= new LinkedList[SZ];
  public Object put(Object key, Object value) {
    Object result = null;
    int index = key.hashCode() % SZ;
    if(index < 0) index = -index;
    if(bucket[index] == null)
      bucket[index] = new LinkedList();
    LinkedList pairs = bucket[index];
    MPair pair = new MPair(key, value);
    ListIterator it = pairs.listIterator();
    boolean found = false;
    while(it.hasNext()) {
      Object iPair = it.next();
      if(iPair.equals(pair)) {
        result = ((MPair)iPair).getValue();
        it.set(pair); // Replace old with new
        found = true;
        break;
      }
    }
    if(!found)
      bucket[index].add(pair);
    return result;
  }
  public Object get(Object key) {
    int index = key.hashCode() % SZ;
    if(index < 0) index = -index;
    if(bucket[index] == null) return null;
    LinkedList pairs = bucket[index];
    MPair match = new MPair(key, null);
```

```
    ListIterator it = pairs.listIterator();
    while(it.hasNext()) {
      Object iPair = it.next();
      if(iPair.equals(match))
        return ((MPair)iPair).getValue();
    }
    return null;
  }
  public Set entrySet() {
    Set entries = new HashSet();
    for(int i = 0; i < bucket.length; i++) {
      if(bucket[i] == null) continue;
      Iterator it = bucket[i].iterator();
      while(it.hasNext())
        entries.add(it.next());
    }
    return entries;
  }
  public static void main(String[] args) {
    SimpleHashMap m = new SimpleHashMap();
    Collections2.fill(m,
      Collections2.geography, 25);
    System.out.println(m);
  }
} ///:~
```

由於 hash table 中的 "slots" 往往被稱爲 *buckets*，所以用來表示實際 table 的那個 array 便被我命名爲 **bucket**。爲了得到平均分佈，buckets 的個數通常是個質數。請注意，它是個 **LinkedList** array，它能夠自動解決碰撞問題 — 每個新加入的物件都會被直接置於 list 尾端。

當指定的 *key* 已在 list 之中，**put()** 便回傳該 *key* 所關聯的舊 *value*，反之則回傳 **null**。程式以 **result** 接收回傳值；**result** 的初値爲 **null**，但如果在 list 中找到指定的 *key*，該 *key* 便會被指派給 **result**。

對 **put()** 和 **get()** 來說，其第一個動作都是呼叫 *key* 的 **hashCode()**，而其結果會被強迫轉爲正數。接著再以此值對 array 的容量取餘數，藉以使其成爲 **bucket** array 的索引值。如果計算出來的位置是 **null**，便表示沒有任何元素被 "hash to" 該位置上，於是便產生一個新的 **LinkedList**，儲

存此次加入的物件。不過，正常程序下應該走訪整個 list，檢查其中是否有重複的 *value*。如果已經存在，舊 *value* 便被置於 **result** 中，而以新 *value* 取代舊 *value*。旗標 **found** 會記錄是否找到舊的 *key-value* pair，如果沒有找到，新的 pair 便會被附加到 list 尾端。

你可以在 **get()** 中看到和 **put()** 相似的碼，但更為簡單。先是計算 **bucket** array 的索引值，如果 **LinkedList** 已經存在的話，就進行搜尋，找出符合者。

entrySet() 必須尋找、走訪所有 list，並將所有元素置於所產生的 **Set** 內。一旦完成，便可將測試用的 *values* 填入 **Map** 之中，並將它們印出。

HashMap 的效能因子

為了探討這個問題，先得介紹幾個必要的術語：

Capacity：容量，表格中的 buckets 數量。

Initial capacity：最初容量，表格建立之初的 buckets 數量。

HashMap 和 **HashSet**：各有建構式，允許你指定最初容量。

Size：大小，表格內目前所有的條目（entries）。

Load factor：負載因子，size/capacity（大小/容量）。負載因子為 0 者，表示一個空表格，0.5 是一個半滿表格，依此類推。一個輕負載表格出現碰撞（collisions）的機會比較低，比較適合安插和搜尋（但是會降低「透過迭代器巡訪」的速度）。**HashMap** 和 **HashSet** 各有建構式允許你指定負載因子，那意味當這個負載因子達到了，容器的容量（buckets 個數）會自動擴充 — 大約成倍擴充，並將原有的物件重新導入新的 buckets 內（這稱為 *rehashing*）。

HashMap 預設的負載因子值是 0.75（意思是除非表格已塞滿 3/4 以上，否則不會進行 rehashing）。這似乎是在時間成本和空間成本上的極佳權衡結果。較高的負載因子能夠降低表格所需空間，但卻增加搜尋上的時間成本。這一點很重要，因為大多數時候你都是在進行搜尋（包括 **get()** 和 **put()**）。

如果你已經知道自己會將許多條目（entries）儲存於 **HashMap**，那麼產生 **HashMap** 時請給定夠大的初始容量，如此便能防止因為 rehashing 而帶來的額外負擔。

覆寫 hashCode()

現在，你已經了解 **HashMap** 函式牽涉了哪些東西，撰寫 **hashCode()** 的種種議題對你會更具意義。

首先，你無法控制 buckets array 的索引值的產生。這個索引值和特定的 **HashMap** 物件的容量有關，而容量和容器的負載情形、負載因子都有關係。你的 **hashCode()** 所產生的值會被進一步處理以產生 bucket 索引（在 **SimpleHashMap** 中，其計算方式只是對 bucket array 的元素個數取餘數）。

設計 **hashCode()** 時最重要的因素就是：不論 **hashCode()** 何時被呼叫，針對同一個物件，每次被呼叫都應該產生相同的值。如果某個物件被 **put()** 置入 **HashMap** 時透過 **hashCode()** 獲得一個值，**get()** 時卻獲得另一個值，那麼你將無法取出該物件。所以如果你的 **hashCode()** 和「物件內可能改變的資料」有關，你必須告訴你的使用者，一旦資料被改變了，將會因為「產生不同的 **hashCode()**」而得出不同的 *key*。

此外，你或許不會想要根據獨一無二的物件資訊來產生 **hashCode()**，尤其像 **this** 這樣的值更是個差勁的 **hashCode()**。因為如果這麼做，你就無法產生一份新的 *key* 並使它和「傳入 **put()** 的原始 *key-value* pair」中的 *key* 相同。這便是發生於 **SpringDetector.java** 的問題，因為 **hashCode()** 的預設實作方法會使用物件的位址。你應該使用物件內某種有意義的識別資訊。

讓我們以 **String** class 為例。**Strings** 具有某個特質，那就是當程式具有多個內容相同的 **String** 物件時，這些 **String** 物件會對應到同一塊位址（詳見附錄 A 對此機制的說明。譯註：此即所謂 *reference counting*）。所以「兩個 **new String("hello")** 所獲得的實體，其所產生的 **hashCode()** 相同」是有道理的。你可以執行以下程式觀察這種現象：

```
//: c09:StringHashCode.java
public class StringHashCode {
  public static void main(String[] args) {
    System.out.println("Hello".hashCode());
    System.out.println("Hello".hashCode());
  }
} ///:~
```

想讓這個程式正確執行，**String** 的 **hashCode()** 必須根據 **String** 的內容來計算才行。

所以，對於一個實用的 **hashCode()** 來說，它必須快速而又有意義。也就是說它必須能夠依據物件的內容來產生 hash code。請記住，此值不見得必須獨一無二（你應該注重速度更甚於唯一性）但是透過 **hashCode()** 和 **equals()**，必須能夠完全決定物件的身份。

由於產生 bucket 索引值之前 **hashCode()** 會更進一步被處理，所以其值所處的範圍並不重要，只要是個 **int** 就行。

這裡還有另一個影響因素：好的 **hashCode()** 所產生的值應該均勻分佈。如果這些數值有密集（cluster）現象，**HashMap** 或 **HashSet** 的某個部位的負載就會很重，比不上擁有均勻分佈能力之 hashing function 來得快速。

以下便是依循上述原則所寫成的範例：

```
//: c09:CountedString.java
// Creating a good hashCode().
import java.util.*;

public class CountedString {
  private String s;
  private int id = 0;
  private static ArrayList created =
    new ArrayList();
  public CountedString(String str) {
    s = str;
    created.add(s);
    Iterator it = created.iterator();
```

```
    // Id is the total number of instances
    // of this string in use by CountedString:
    while(it.hasNext())
      if(it.next().equals(s))
        id++;
  }
  public String toString() {
    return "String: " + s + " id: " + id +
      " hashCode(): " + hashCode() + "\n";
  }
  public int hashCode()
    return s.hashCode() * id;
  }
  public boolean equals(Object o) {
    return (o instanceof CountedString)
      && s.equals(((CountedString)o).s)
      && id == ((CountedString)o).id;
  }
  public static void main(String[] args) {
    HashMap m = new HashMap();
    CountedString[] cs = new CountedString[10];
    for(int i = 0; i < cs.length; i++) {
      cs[i] = new CountedString("hi");
      m.put(cs[i], new Integer(i));
    }
    System.out.println(m);
    for(int i = 0; i < cs.length; i++) {
      System.out.print("Looking up " + cs[i]);
      System.out.println(m.get(cs[i]));
    }
  }
} ///:~
```

CountedString 內含一個 **String** 以及一個用以表示編號的 **id**。所有 **CountedString** 物件都內含一模一樣的 **String**。所有 **String**s 都被儲存於 **static ArrayList** 內，建構式透過走訪 **ArrayList** 完成元素個數的計算動作。

hashCode() 和 **equals()** 都是根據兩個資料成員來求結果；如果它們只根據 **String** 或只根據 **id**，那麼不同的 *value*s 就有可能被視爲相同（那當然就不妙了）。

請注意 **hashCode()** 多麼簡單：將 **String** 的 **hashCode()** 乘以 **id**。對 **hashCode()** 來說，愈小通常代表愈好（而且也愈快）。

main() 產生了許多 **CountedString** 物件，並使用相同的 **String**，藉以說明「雖然 **String** 的內容相同，但由於 **id** 不同，所以產生的 **hashCode()** 也不相同」。程式會顯示 **HashMap()** 內容，你可以從中觀察其儲存情形（但不具任何可識別的次序），然後便是個別搜尋每個 *key*，藉以說明搜尋機制的確運作無誤。

持有 references

java.lang.ref 程式庫含有一組 classes，這些 classes 爲垃圾回收機制提供更大的彈性。當你擁有許多可能耗盡記憶體的大型物件時，這些 classes 格外有用。有三個 classes 繼承自抽象類別 **Reference**：**SoftReference, WeakReference** 和 **PhantomReference**。它們都對垃圾回收機制提供不同層次的間接性 — 如果外界只能透過這些 **Reference** 物件才得碰觸物件的話。

某個物件如果是「可碰觸的（*reachable*）」，代表程式中的某處可以找到該物件。這意味你可能擁有 stack 上的某個一般性 reference，它恰好指向某個物件；但也可能你擁有一個 reference 指向某物件，而該物件擁有一個 reference，指向討論中的這個物件；這中間可能存在許多中介鏈結（*intermediate links*）。如果物件是可碰觸的，垃圾回收器就不能加以釋放（回收），因爲它仍然被程式所用。如果某個物件不可碰觸，你的程式便無法使用它，那麼垃圾回收器便可安全地將它回收。

當你希望繼續持有某個 reference（它所指的物件是你希望碰觸的），但你也允許垃圾回收器將它釋放（回收），這時候你應該使用 **Reference** 物件。於是你得以繼續使用該物件，而一旦記憶體即將耗盡，你也允許它被釋放。

以 **Reference** 物件做爲你和一般 reference 之間的中介，便可達到上述目的。注意，不能再有其他一般 reference 指向那個物件（那是一個未被包裝（wrapped）於 **Reference** 物件內的物件）。如果垃圾回收器發現某個物件可透過一般 reference 被碰觸，就不會釋放該物件。

一共有三種 **Reference**s。依照 **SoftReference**, **WeakReference**, **PhantomReference** 的次序，每一個都弱於後者，並且分別對應於不同等級的可碰觸能力。*Soft* references 乃是爲了實作出與記憶體極端相依的快取裝置（memory-sensitive cache）而設計，*Weak* references 是爲了實作所謂標準映射（canonicalizing mapping）而設計（於是物件實體可同時被使用於程式多處以節省儲存空間），它並不會妨礙物件的 *key*s（或 *value*s）被重新宣告。*Phantom* references 是爲了以一種比 Java 終止機制（finalization mechanism）更具彈性的方式來對 "pre-mortem" 清理動作進行排程（scheduling）。

使用 **SoftReference**s 和 **WeakReference**s 時，你可以選擇是否要將它們置於一個 **ReferenceQueue** 中（這是一種用於 premortem 清理動作的工具）。但當你使用 **PhantomReference**s 時，你只能將它們建於一個 **ReferenceQueue** 內。下面是一個簡單的例子：

```
//: c09:References.java
// Demonstrates Reference objects
import java.lang.ref.*;

class VeryBig {
  static final int SZ = 10000;
  double[] d = new double[SZ];
  String ident;
  public VeryBig(String id) { ident = id; }
  public String toString() { return ident; }
  public void finalize() {
    System.out.println("Finalizing " + ident);
  }
}

public class References {
  static ReferenceQueue rq= new ReferenceQueue();
  public static void checkQueue() {
    Object inq = rq.poll();
```

```
    if(inq != null)
      System.out.println("In queue: " +
        (VeryBig)((Reference)inq).get());
  }
  public static void main(String[] args) {
    int size = 10;
    // Or, choose size via the command line:
    if(args.length > 0)
      size = Integer.parseInt(args[0]);
    SoftReference[] sa =
      new SoftReference[size];
    for(int i = 0; i < sa.length; i++) {
      sa[i] = new SoftReference(
        new VeryBig("Soft " + i), rq);
      System.out.println("Just created: " +
        (VeryBig)sa[i].get());
      checkQueue();
    }
    WeakReference[] wa =
      new WeakReference[size];
    for(int i = 0; i < wa.length; i++) {
      wa[i] = new WeakReference(
        new VeryBig("Weak " + i), rq);
      System.out.println("Just created: " +
        (VeryBig)wa[i].get());
      checkQueue();
    }
    SoftReference s = new SoftReference(
      new VeryBig("Soft"));
    WeakReference w = new WeakReference(
      new VeryBig("Weak"));
    System.gc();
    PhantomReference[] pa =
      new PhantomReference[size];
    for(int i = 0; i < pa.length; i++) {
      pa[i] = new PhantomReference(
        new VeryBig("Phantom " + i), rq);
      System.out.println("Just created: " +
        (VeryBig)pa[i].get());
      checkQueue();
    }
  }
} ///:~
```

一旦執行這個程式（建議你將輸出結果導至 "more" 工具程式，這麼一來就可以一頁一頁地觀看輸出結果），你會看到物件已被垃圾回收器回收 — 即使你仍可透過 **Reference** 物件加以存取（如果要取得實際的 object reference，可使用 **get()**）。你也會看到 **ReferenceQueue** 總是產生「內含一個 **null** 物件」的 **Reference**。爲了善用此一機制，你可以繼承你感興趣的某個 **Reference** class，並爲你的這個新的 **Reference** 型別加入更多有用的函式。

WeakHashMap

容器庫中有一個特殊的 **Map**，其所持有的乃是 weak references：**WeakHashMap**。這個 class 被設計用來產生所謂的 *canonicalized mappings*（譯註：見前一小節）。在這樣一個 mapping 中，由於每個 *value* 只存在單一實體，因而節省了儲存空間。一旦程式需要某個 *value*，便在 mapping 中搜尋既有的物件，並使用找到的那個物件（而非重新再造一個）。mapping 可能會在初始化的時候便產生所含之值，但更有可能在必要的時候才產生。

由於這是一種節省儲存空間的技巧，所以 **WeakHashMap** 能夠方便地讓垃圾回收器自動清理 *key* 和 *value*。你不需要對你打算置入 **WeakHashMap** 內的 *key* 和 *value* 做什麼特殊動作。一旦 *key* 不再被使用，便會觸發清理動作，如下所示：

```
//: c09:CanonicalMapping.java
// Demonstrates WeakHashMap.
import java.util.*;
import java.lang.ref.*;

class Key {
  String ident;
  public Key(String id) { ident = id; }
  public String toString() { return ident; }
  public int hashCode()
    return ident.hashCode();
  }
```

```
  public boolean equals(Object r) {
    return (r instanceof Key)
      && ident.equals(((Key)r).ident);
  }
  public void finalize() {
    System.out.println("Finalizing Key "+ ident);
  }
}

class Value {
  String ident;
  public Value(String id) { ident = id; }
  public String toString() { return ident; }
  public void finalize() {
    System.out.println("Finalizing Value "+ident);
  }
}

public class CanonicalMapping {
  public static void main(String[] args) {
    int size = 1000;
    // Or, choose size via the command line:
    if(args.length > 0)
      size = Integer.parseInt(args[0]);
    Key[] keys = new Key[size];
    WeakHashMap whm = new WeakHashMap();
    for(int i = 0; i < size; i++) {
      Key k = new Key(Integer.toString(i));
      Value v = new Value(Integer.toString(i));
      if(i % 3 == 0)
        keys[i] = k; // Save as "real" references
      whm.put(k, v);
    }
    System.gc();
  }
} ///:~
```

正如本章先前所討論的，**Key** class 必須具備 **hashCode()** 和 **equals()**，因爲它被用來做爲一個 hashed 資料結構中的 *key*。

當你執行這個程式，你會發現，垃圾回收器每經三個 *keys* 便略過一個，因爲指向該 *key* 的一般性 reference 已經被置於 **keys** array 內，因此這些物件不會被垃圾回收器回收。

再論 Iterators（迭代器）

現在我們可以說明 **Iterator** 的實際威力了：它能夠將序列走訪能力和序列底層結構隔離開來。下例中的 class **PrintData** 會使用 **Iterator** 來走訪序列，並呼叫每個物件的 **toString()**。這裡會產生兩個不同型態的容器：**ArrayList** 和 **HashMap**，分別擁有 **Mouse** 物件和 **Hamster** 物件（本章先前已定義過這兩個 classes）。由於 **Iterator** 會將容器的底層結構隱藏起來，所以 **PrintData** 不知道也不在乎 **Iterator** 來自哪一種容器：

```
//: c09:Iterators2.java
// Revisiting Iterators.
import java.util.*;

class PrintData {
  static void print(Iterator e) {
    while(e.hasNext())
      System.out.println(e.next());
  }
}

class Iterators2 {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 5; i++)
      v.add(new Mouse(i));
    HashMap m = new HashMap();
    for(int i = 0; i < 5; i++)
      m.put(new Integer(i), new Hamster(i));
    System.out.println("ArrayList");
    PrintData.print(v.iterator());
    System.out.println("HashMap");
    PrintData.print(m.entrySet().iterator());
  }
```

```
} ///:~
```

對 **HashMap** 而言，**entrySet()** 會產生一個 **Set**，內含 **Map.entry** 物件。後者含有每個元素的 *key* 和 *value*，所以你會看到兩者皆被印出。

請注意，**PrintData.print()** 善用了「容器內的物件皆隸屬 **Object**」這個特性，使 **System.out.println()** 自動呼叫 **toString()**。面對你自己的問題時，很可能你得假設 **Iterator** 所走訪的乃是某個特定型別的容器。例如你可能得假設，容器內的所有東西都是「擁有 **draw()** 函式」的 **Shape**。然後你就必須將 **Iterator.next()** 回傳的 **Object** 向下轉型至 **Shape**。

選擇一份適當的實作品

Choosing an implementation

現在，你應該已經了解，實際上只有三種容器組件：**Map**, **List**, **Set**，而且每個 interface 都只有二至三種實作品。當你需要使用某個 interface 所提供的功能時，該如何選擇呢？

欲了解這個問題的答案，你必須先認識清楚，不同的實作有其不同的特性、優點和缺點。你依舊可以看到 **Hashtable**, **Vector**, **Stack** 的特性，它們都是舊的容器，所以先前撰寫的程式碼不會無從運作。如果你不打算使用新的（Java 2 提供的）容器，它們的確是最好的。

其他容器間的差別，取決於其背後提供支援的結構 — 亦即實際用以實作出「我們所欲使用之 interface」的資料結構。舉個例子，**ArrayList** 和 **LinkedList** 都實作了 **List** interface，所以不論使用何者，程式都會產生相同結果。

然而 **ArrayList** 的底層以 array 完成，**LinkedList** 則是以一般的雙向串列（double-linked list）完成，其內每個物件除了資料本身外，還有兩個 references，分別指向前一個元素和後一個元素。因此如果你可能在 list 中央進行許多安插或移除動作，**LinkedList** 比較適當（此外 **LinkedList** 還具備一些由 **AbstractSequentialList** 建構出來的額外功能）。如果並非如此，那麼 **ArrayList** 通常比較快。

再舉一個例子。**Set** 可被實作爲 **TreeSet** 或 **HashSet**。**TreeSet** 的底層是 **TreeMap**，被設計用來產生一個始終依序排列的集合。不過如果你的 **Set** 之中有大量物件，那麼 **TreeSet** 的安插效率便會變得緩慢。當你撰寫一個會用到 **Set** 的程式時，應該先考慮使用 **HashSet**，只有在「保持集合內各元素的排序狀態」極爲重要時，才選擇 **TreeSet**。

在各種 Lists 之間抉擇

如果想知道各種 **Lists** 之間的差異，最方便的辦法就是來一次效能測試。以下程式碼會建立一個 inner base class，做爲測試框架，然後產生一個由匿名 inner classes 組成的 array，每一個 inner class 用於一種測試。每一個 inner class 都會被 **test()** 呼叫。這個作法讓你得以輕鬆增加新型式的測試，或移除某種測試。

```
//: c09:ListPerformance.java
// Demonstrates performance differences in Lists.
import java.util.*;
import com.bruceeckel.util.*;

public class ListPerformance {
  private abstract static class Tester {
    String name;
    int size; // Test quantity
    Tester(String name, int size)
      this.name = name;
      this.size = size;
    }
    abstract void test(List a, int reps);
  }
```

```
  private static Tester[] tests = {
    new Tester("get", 300) {
      void test(List a, int reps) {
        for(int i = 0; i < reps; i++) {
          for(int j = 0; j < a.size(); j++)
            a.get(j);
        }
      }
    },
    new Tester("iteration", 300) {
      void test(List a, int reps) {
        for(int i = 0; i < reps; i++) {
          Iterator it = a.iterator();
          while(it.hasNext())
            it.next();
        }
      }
    },
    new Tester("insert", 5000) {
      void test(List a, int reps) {
        int half = a.size()/2;
        String s = "test";
        ListIterator it = a.listIterator(half);
        for(int i = 0; i < size * 10; i++)
          it.add(s);
      }
    },
    new Tester("remove", 5000) {
      void test(List a, int reps) {
        ListIterator it = a.listIterator(3);
        while(it.hasNext()) {
          it.next();
          it.remove();
        }
      }
    },
  };
  public static void test(List a, int reps) {
    // A trick to print out the class name:
    System.out.println("Testing " +
      a.getClass().getName());
```

```
    for(int i = 0; i < tests.length; i++) {
      Collections2.fill(a,
        Collections2.countries.reset(),
        tests[i].size);
      System.out.print(tests[i].name);
      long t1 = System.currentTimeMillis();
      tests[i].test(a, reps);
      long t2 = System.currentTimeMillis();
      System.out.println(": " + (t2 - t1));
    }
  }
  public static void testArray(int reps) {
    System.out.println("Testing array as List");
    // Can only do first two tests on an array:
    for(int i = 0; i < 2; i++) {
      String[] sa = new String[tests[i].size];
      Arrays2.fill(sa,
        Collections2.countries.reset());
      List a = Arrays.asList(sa);
      System.out.print(tests[i].name);
      long t1 = System.currentTimeMillis();
      tests[i].test(a, reps);
      long t2 = System.currentTimeMillis();
      System.out.println(": " + (t2 - t1));
    }
  }
  public static void main(String[] args) {
    int reps = 50000;
    // Or, choose the number of repetitions
    // via the command line:
    if(args.length > 0)
      reps = Integer.parseInt(args[0]);
    System.out.println(reps + " repetitions");
    testArray(reps);
    test(new ArrayList(), reps);
    test(new LinkedList(), reps);
    test(new Vector(), reps);
  }
} ///:~
```

inner class **Tester** 被宣告爲 **abstract**，做爲各種測試的 base class。它內含一個 **String**（用以在測試開始時列印出來）、一個 **size** 參數（用以表示測試的元素個數或測試動作的重複次數）、一個建構式（用以初始化各個資料成員）、一個 **abstract test()**（用以執行實際測試工作）。所有不同類型的測試動作都被集中置於 **tests** array 內。程式會將各個繼承自 **Tester** 的匿名 inner classes 設爲此一 array 的初始值。如果想增加或移除測試動作，只要將 inner class 的定義加至此 array 內，或是自 array 中移除，其餘所有必要動作便會自動進行。

爲了比較 array 和容器（主要是 **ArrayList**）的存取，這裡有個特殊測試動作，會利用 **Arrays.asList()** 將 array 包裝爲一個 **List**。請注意，在此情形下只有前兩個測試可以執行，因爲我們無法對 array 執行安插或移除（元素）的動作。

test() 接到它將處理的 **List** 之後，首先爲後者充填元素，然後記錄 **tests** array 內每個測試動作的執行時間。記錄所得會因機器設備的不同而有差異。這些結果只是爲了比較不同容器的執行效能的相對差異。以下便是某次執行的結果：

容器型別	取得 Get	迭代 Iteration	安插 Insert	移除 Remove
array	1430	3850	na	na
ArrayList	3070	12200	500	46850
LinkedList	16320	9110	110	60
Vector	4890	16250	550	46850

一如預期，array 在隨機搜尋和迭代動作上都比其他容器快。你可以看到，隨機存取動作（**get()**）對 **ArrayLists** 而言代價並不高，但對 **LinkedList** 來說卻是代價高昂。奇怪的是 **LinkedList** 的迭代速度比 **ArrayList** 快，這有點違反直覺。另一方面，list 中心處的安插和移除動作，**LinkedList** 比 **ArrayList** 快多了，尤其是移除動作。**Vector** 通常不像 **ArrayList** 那麼快，因此應該避免使用；它存在於容器庫中只不過是爲了兼容老舊程式碼；它能夠在上述程式中運作，完全是因爲它在 Java2

中被轉換為 **List**。最好的作法可能是以 **ArrayList** 做為你的預設選擇，當你發現效能問題出於「在 list 中心處進行過多的安插和移除動作」，才轉而使用 **LinkedList**。當然，如果你所處理的是固定數量的一組元素，請使用 array。

在各種 Sets 之間抉擇

你可以依據 **Set** 的大小，選擇使用 **TreeSet** 或 **HashSet**（但如果你需要產生一個經過排序的序列，請使用 **TreeSet**）。以下測試程式說明了其間的取捨權衡：

```
//: c09:SetPerformance.java
import java.util.*;
import com.bruceeckel.util.*;

public class SetPerformance {
  private abstract static class Tester {
    String name;
    Tester(String name) { this.name = name; }
    abstract void test(Set s, int size, int reps);
  }
  private static Tester[] tests = {
    new Tester("add")
      void test(Set s, int size, int reps) {
        for(int i = 0; i < reps; i++) {
          s.clear();
          Collections2.fill(s,
            Collections2.countries.reset(),size);
        }
      }
    },
    new Tester("contains")
      void test(Set s, int size, int reps) {
        for(int i = 0; i < reps; i++)
          for(int j = 0; j < size; j++)
            s.contains(Integer.toString(j));
      }
    },
    new Tester("iteration")
```

```
      void test(Set s, int size, int reps) {
        for(int i = 0; i < reps * 10; i++) {
          Iterator it = s.iterator();
          while(it.hasNext())
            it.next();
        }
      }
    },
  };
  public static void
  test(Set s, int size, int reps) {
    System.out.println("Testing " +
      s.getClass().getName() + " size " + size);
    Collections2.fill(s,
      Collections2.countries.reset(), size);
    for(int i = 0; i < tests.length; i++) {
      System.out.print(tests[i].name);
      long t1 = System.currentTimeMillis();
      tests[i].test(s, size, reps);
      long t2 = System.currentTimeMillis();
      System.out.println(": " +
        ((double)(t2 - t1)/(double)size));
    }
  }
  public static void main(String[] args) {
    int reps = 50000;
    // Or, choose the number of repetitions
    // via the command line:
    if(args.length > 0)
      reps = Integer.parseInt(args[0]);
    // Small:
    test(new TreeSet(), 10, reps);
    test(new HashSet(), 10, reps);
    // Medium:
    test(new TreeSet(), 100, reps);
    test(new HashSet(), 100, reps);
    // Large:
    test(new TreeSet(), 1000, reps);
    test(new HashSet(), 1000, reps);
  }
} ///:~
```

下表列出某次執行的結果。注意，執行結果會隨著硬體設備以及你所使用的 Java 虛擬機器（JVM）而有不同。你應該自己執行一下這個測試程式。

型別	測試大小	**Add**	**Contains**	**Iteration**
	10	138.0	115.0	187.0
TreeSet	100	189.5	151.1	206.5
	1000	150.6	177.4	40.04
	10	55.0	82.0	192.0
HashSet	100	45.6	90.0	202.2
	1000	36.14	106.5	39.39

對所有動作而言，**HashSet** 的效能通常都優於 **TreeSet**（尤其是最重要的兩個動作：安插和搜尋）。**TreeSet** 存在的唯一理由是：它能夠維護其內元素的排序狀態。所以，只有在你需要這個性質時你才應該使用它。

在各種 Maps 之間抉擇

Maps 的大小深深影響其效能表現。以下測試程式說明了取捨權衡：

```
//: c09:MapPerformance.java
// Demonstrates performance differences in Maps.
import java.util.*;
import com.bruceeckel.util.*;

public class MapPerformance {
  private abstract static class Tester {
    String name;
    Tester(String name) { this.name = name; }
    abstract void test(Map m, int size, int reps);
  }
  private static Tester[] tests = {
    new Tester("put")
      void test(Map m, int size, int reps) {
        for(int i = 0; i < reps; i++) {
          m.clear();
          Collections2.fill(m,
```

```
          Collections2.geography.reset(), size);
        }
      }
    },
    new Tester("get") {
      void test(Map m, int size, int reps) {
        for(int i = 0; i < reps; i++)
          for(int j = 0; j < size; j++)
            m.get(Integer.toString(j));
      }
    },
    new Tester("iteration") {
      void test(Map m, int size, int reps) {
        for(int i = 0; i < reps * 10; i++) {
          Iterator it = m.entrySet().iterator();
          while(it.hasNext())
            it.next();
        }
      }
    },
  };
  public static void
  test(Map m, int size, int reps) {
    System.out.println("Testing " +
      m.getClass().getName() + " size " + size);
    Collections2.fill(m,
      Collections2.geography.reset(), size);
    for(int i = 0; i < tests.length; i++) {
      System.out.print(tests[i].name);
      long t1 = System.currentTimeMillis();
      tests[i].test(m, size, reps);
      long t2 = System.currentTimeMillis();
      System.out.println(": " +
        ((double)(t2 - t1)/(double)size));
    }
  }
  public static void main(String[] args) {
    int reps = 50000;
    // Or, choose the number of repetitions
    // via the command line:
    if(args.length > 0)
```

```
      reps = Integer.parseInt(args[0]);
    // Small:
    test(new TreeMap(), 10, reps);
    test(new HashMap(), 10, reps);
    test(new Hashtable(), 10, reps);
    // Medium:
    test(new TreeMap(), 100, reps);
    test(new HashMap(), 100, reps);
    test(new Hashtable(), 100, reps);
    // Large:
    test(new TreeMap(), 1000, reps);
    test(new HashMap(), 1000, reps);
    test(new Hashtable(), 1000, reps);
  }
} ///:~
```

由於 **Map** 的大小是重要影響因素，你可以看到，這份測試所量測出來的時間，除以測試大小，會使每次量測呈現常態化（normalized）。以下是一組結果（你的結果可能有所不同）。

型別	測試大小	**Put**	**Get**	**Iteration**
TreeMap	10	143.0	110.0	186.0
	100	201.1	188.4	280.1
	1000	222.8	205.2	40.7
HashMap	10	66.0	83.0	197.0
	100	80.7	135.7	278.5
	1000	48.2	105.7	41.4
Hashtable	10	61.0	93.0	302.0
	100	90.6	143.3	329.0
	1000	54.1	110.95	47.3

一如你所預期，**Hashtable** 的效率大致上和 **HashMap** 相同（你也看到了，**HashMap** 一般來說稍微快些。的確，**HashMap** 是 **Hashtable** 的替代品）。**TreeMap** 通常比 **HashMap** 慢，那麼何必使用它呢？因爲你可以不把它當做 **Map**，而把它當做一種產生有序串列（ordered list）的手

法。tree 特性使得它總是保有某種次序，但不見得有什麼特定排序。一旦你將元素置入 **TreeMap**，便可以呼叫 **keySet()** 取得一份以 **Set** 為觀點的 *keys*，然後可以呼叫 **toArray()** 產生這些 *keys* 所組成的 array。接下來便可使用 **static Array.binarySearch()**（稍後討論）於排序狀態下的 array 內進行快速搜尋。當然，如果基於某種理由，你只想執行這樣的動作，那麼就不適合採用 **HashMap**，因為 **HashMap** 被設計用於快速搜尋。你可以輕易透過單一物件的生成，利用 **TreeMap** 產生一個 **HashMap**。結論是，當你使用 **Map**，應該優先選擇 **HashMap**；只有當你的 **Map** 必須持續保持排序狀態，才需要動用 **TreeMap**。

Lists 的排序和搜尋

對 **Lists** 進行排序和搜尋的那些公用函式（utilities），和用於 *objects array* 身上的一樣，具有相同的名稱和標記式（signatures）。只不過這些 **static** 函式隸屬於 **Collections** 而非 **Arrays**。以下是個例子，修改自 **ArraySearch.java**：

```
//: c09:ListSortSearch.java
// Sorting and searching Lists with 'Collections.'
import com.bruceeckel.util.*;
import java.util.*;

public class ListSortSearch {
  public static void main(String[] args) {
    List list = new ArrayList();
    Collections2.fill(list,
      Collections2.capitals, 25);
    System.out.println(list + "\n");
    Collections.shuffle(list);
    System.out.println("After shuffling: "+list);
    Collections.sort(list);
    System.out.println(list + "\n");
    Object key = list.get(12);
    int index =
      Collections.binarySearch(list, key);
    System.out.println("Location of " + key +
```

```
      " is " + index + ", list.get(" +
      index + ") = " + list.get(index));
    AlphabeticComparator comp =
      new AlphabeticComparator();
    Collections.sort(list, comp);
    System.out.println(list + "\n");
    key = list.get(12);
    index =
      Collections.binarySearch(list, key, comp);
    System.out.println("Location of " + key +
      " is " + index + ", list.get(" +
      index + ") = " + list.get(index));
  }
} ///:~
```

這些函式的使用方式，和應用於 **Array**s 身上的一樣。只不過你現在用的是一個 **List** 而非一個 array。和 array 的情況相同，如果你使用 **Comparator** 來進行排序，你就得使用同樣的 **Comparator** 來進行 **binarySearch()**。

這個程式同時也示範了 **Collection**s 中的 **shuffle()**，它能以隨機方式改變 **List** 的次序（order）。

公用函式（Utilities）

Collections class 提供了一些有用的工具：

enumeration(Collection)	爲引數產生一個舊式的 **Enumeration**。
max(Collection) **min(Collection)**	找出（並傳回）引數中的最大或最小元素 – 採用 **Collection** 內含之物件的自然比較法（natural comparison）。
max(Collection, Comparator) **min(Collection, Comparator)**	使用 **Comparator**，回傳 **Collection** 內的最大或最小元素。
reverse( )	將所有元素反轉於適當位置。

copy(List dest, List src)	將 src 中的元素複製到 dest。
fill(List list, Object o)	以 o 取代 list 內的所有元素。
nCopies(int n, Object o)	回傳一個「大小為 n、內容不再更動」的 **List**，其中所有 reference 皆指向 o。

請注意，**min()** 和 **max()** 都能處理 **Collection** 物件，但不能處理 **List**。所以你不需要擔心 **Collection** 是否應該先經過排序。一如先前所說，只有在執行 **binarySearch()** 之前，你才需要先對 **List** 或 array 做 **sort()** 動作。

讓 Collection 或 Map 無法被更改

通常，撰寫唯讀版本的 **Collection** 或 **Map**，可能為你帶來許多便利。**Collections** class 允許你只要將原來的容器傳入某個函式中便可得到一份唯讀版本。這個函式共有四種變形：分別用於 **Collection**（如果你並不想將 **Collection** 視為更特定的型別的話）、**List**、**Set**、**Map**。以下例子示範建構各種型別之唯讀版本的正確方式：

```
//: c09:ReadOnly.java
// Using the Collections.unmodifiable methods.
import java.util.*;
import com.bruceeckel.util.*;

public class ReadOnly {
  static Collections2.StringGenerator gen =
    Collections2.countries;
  public static void main(String[] args) {
    Collection c = new ArrayList();
    Collections2.fill(c, gen, 25); // Insert data
    c = Collections.unmodifiableCollection(c);
    System.out.println(c); // Reading is OK
    c.add("one"); // Can't change it

    List a = new ArrayList();
    Collections2.fill(a, gen.reset(), 25);
```

```
    a = Collections.unmodifiableList(a);
    ListIterator lit = a.listIterator();
    System.out.println(lit.next()); // Reading OK
    lit.add("one"); // Can't change it

    Set s = new HashSet();
    Collections2.fill(s, gen.reset(), 25);
    s = Collections.unmodifiableSet(s);
    System.out.println(s); // Reading OK
    //! s.add("one"); // Can't change it

    Map m = new HashMap();
    Collections2.fill(m,
      Collections2.geography, 25);
    m = Collections.unmodifiableMap(m);
    System.out.println(m); // Reading OK
    //! m.put("Ralph", "Howdy!");
  }
} ///:~
```

不論哪一種情況，你都必須在你的容器成為唯讀之前，先將有意義的資料填入。載入資料後，最好的作法就是以「不會造成變動」之呼叫（unmodifiable call）所產生的 reference 替換掉原有的 reference。藉由這個方式，容器獲得唯讀性質之後，你就不可能「不小心改變其內容」了。另一方面，這個手法也讓你得以在 class 內部將「打算被修改的容器」宣告為 **private**，並利用某個函式傳回「指向該容器」的一個唯讀的 reference。於是你就可以在 class 內部改變容器內容，但其他人只能讀取。

針對某個特定型別呼叫一個「不會造成變動」的函式，並不會引起編譯期檢驗。但是一旦轉換發生，所有「會造成容器內容變動」的 method calls 都會引發 **UnsupportedOperationException**。

Collection 或 Map 的同步控制

Synchronizing a **Collection** or **Map**

關鍵字 **synchronized** 在「多緒」（multithreading）課題中是十分重要的一環，本書第 14 章才會導入這個課題。這裡我要先提醒你，只有

Collection class 才具備「自動同步控制整個容器」的機制。其語法和「不會造成變動的」（unmodifiable）函式類似：

```
//: c09:Synchronization.java
// Using the Collections.synchronized methods.
import java.util.*;

public class Synchronization {
  public static void main(String[] args) {
    Collection c =
      Collections.synchronizedCollection(
        new ArrayList());
    List list = Collections.synchronizedList(
      new ArrayList());
    Set s = Collections.synchronizedSet(
      new HashSet());
    Map m = Collections.synchronizedMap(
      new HashMap());
  }
} ///:~
```

此例將新產生的容器立刻傳入適當的 "synchronized" 函式中；透過這種方式，就不會意外產生不同步的版本。

Fail fast（當機立斷判出局）

Java 容器還有一個機制，可以防止多個行程（processes）修改同一個容器的內容。如果你正在走訪某個容器，另一個行程卻要進行元素的安插、刪除、修改等動作，便會發生問題：或許你已經走過了那個物件、或許你還沒有、或許在你呼叫 **size()** 之後容器的大小縮減了…太多太多不幸的情況。Java 容器庫有一個所謂的 *fail-fast* 機制：它會搜尋「因你的行程而做的任何改變」之外的所有容器變化。如果它偵測到其他行程也在修改同一個容器，就會立刻產生一個 **ConcurrentModifictionException**。此即所謂 "fail-fast" — 它並不會試著於稍後才以更複雜的演算法來偵測問題。

很容易就可以觀察到 fail-fast 機制的運作。你唯一需要做的就是產生一個迭代器，然後將某些東西加到迭代器所指的那個 **Collection** 中，像這樣：

```
//: c09:FailFast.java
// Demonstrates the "fail fast" behavior.
import java.util.*;

public class FailFast {
  public static void main(String[] args) {
    Collection c = new ArrayList();
    Iterator it = c.iterator();
    c.add("An object");
    // Causes an exception:
    String s = (String)it.next();
  }
} ///:~
```

這樣便會引發異常，因爲在向容器取得迭代器之後，某些東西又被置入容器內。程式中有兩個地方都有可能修改到同一個容器，並因而產生不確定狀態。引發的那個異常便是通知你說，你應該修改你的程式 – 本例之中你得在加入所有元素之後，才取迭代器。

請注意，當你運用 **get()** 來存取 **List** 的元素時，無法從此類監看動作中得到好處。

未獲支援的操作（Unsupported operations）

透過 **Arrays.asList()**，我們可以將 array 轉換爲 **List**：

```
//: c09:Unsupported.java
// Sometimes methods defined in the
// Collection interfaces don't work!
import java.util.*;

public class Unsupported {
  private static String[] s = {
    "one", "two", "three", "four", "five",
    "six", "seven", "eight", "nine", "ten",
```

```
  };
  static List a = Arrays.asList(s);
  static List a2 = a.subList(3, 6);
  public static void main(String[] args) {
    System.out.println(a);
    System.out.println(a2);
    System.out.println(
      "a.contains(" + s[0] + ") = " +
      a.contains(s[0]));
    System.out.println(
      "a.containsAll(a2) = " +
      a.containsAll(a2));
    System.out.println("a.isEmpty() = " +
      a.isEmpty());
    System.out.println(
      "a.indexOf(" + s[5] + ") = " +
      a.indexOf(s[5]));
    // Traverse backwards:
    ListIterator lit = a.listIterator(a.size());
    while(lit.hasPrevious())
      System.out.print(lit.previous() + " ");
    System.out.println();
    // Set the elements to different values:
    for(int i = 0; i < a.size(); i++)
      a.set(i, "47");
    System.out.println(a);
    // Compiles, but won't run:
    lit.add("X"); // Unsupported operation
    a.clear(); // Unsupported
    a.add("eleven"); // Unsupported
    a.addAll(a2); // Unsupported
    a.retainAll(a2); // Unsupported
    a.remove(s[0]); // Unsupported
    a.removeAll(a2); // Unsupported
  }
} ///:~
```

你會發現，**Collection** 和 **List** interfaces 之中，只有一部份被確確實實地實作出來。如果你呼叫了其他函式，便會引發令人討厭的異常 **UnsupportedOperationException**。下一章你會學到各種異常，這裡

長話短說：**Collection**（或 Java 容器庫內的其他某些 interfaces）內含所謂「可有可無的（optional）」函式，它們有可能在實作該 interface 的具象類別中獲得支援，但也可能不會。如果你呼叫未獲支援的函式，會引發 **UnsupportedOperationException**，表示編程有問題。

『什麼?!』你一定難以置信：『interfaces 和 base classes 的整個重點不就是在於保證這些函式一定會執行某種有意義的動作嗎？上述說法破壞了這個承諾，使得呼叫某些函式時不僅不會執行有意義的行爲，甚至會中斷整個程式的執行！型別的安全性簡直被棄若鄙屣！』

情況其實沒有那麼糟糕。使用 **Collection**、**List**、**Set**、**Map** 時，編譯器會嚴格限制你只能呼叫 interface 內的函式，所以和 Smalltalk 並不一樣（在 Smalltalk 中你可以呼叫任何物件的任何函式，而且只有當你執行你的程式時，你才會知道你的呼叫是否做了些什麼動作）。此外，大多數接受 **Collection** 做爲引數的函式，都只會讀取其內容，而 **Collection** 的所有 "read" 函式都不是「可有可無的」。

這個方法能防止設計過程中的 intefaces 數量大增。其他容器庫似乎都提供了過多令人混淆的 interfaces 來描述主架構的各種變形，因而變得難以學習。由於每個人都可能發明新的 interface，所以甚至不可能找出 **interface**s 的所有特例。「未獲支援的操作」達成了 Java 容器庫的一個重要目標：容器必須易學易用；這些「未獲支援的動作」是一種「可以稍晚再學習」的特殊個案。然而，爲了讓這種作法成立：

1. **UnsupportedOperationException** 必須是個很少出現的事件（event）。也就是說對大多數 classes 而言，所有操作都應該能夠運行才是。只有在特殊情況下才可以不支援某項操作。Java 容器庫亦然，因爲 99%的時間裡頭，你可能使用的 classes（**ArrayList**、**LinkedList**、**HashSet**、**HashMap**，以及其他具象實作類別）都支援所有操作。如果你想要產生新的 **Collection**，但不想爲 **Collection** interface 中的所有函式都提供定義，卻想讓它和既有程式庫整合，這樣的設計便可爲你提供一扇「後門」。

2. 當某個操作不被支援，實作時（而非等到產品出貨給顧客後）就出現 **UnsupportedOperationException**，應該是個很合乎道理的結果。畢竟它代表的是編程上的錯誤：你使用了不正確的實作物（implementation）。事實不見得如此，而這也正是此一設計的實驗性質發揮作用的地方。只有隨著時間過去，我們才會知道它的功效如何。

上例中的 **Arrays.asList()** 會產生一個「以固定容量的 array 爲基底」的 **List**。因此對它而言，僅支援那些「永遠不會改變 array 容量」的操作，是很合理的。如果有個新的 interface 用來表示不同種類的行爲（或許名爲 "**FixedSizeList**"），那麼便會打開複雜度的大門。很快地，在你試著使用程式庫時，你會不知道從何著手。

一個接受 **Collection** 或 **List** 或 **Set** 或 **Map** 爲引數的函式，其說明文件應該指出哪些可有可無的函式一定得被實作出來。例如，排序需要用到 **set()** 和 **Iterator.set()**，卻不需要 **add()** 和 **remove()**。

Java 1.0/1.1 的容器

很不幸的是，許多老舊程式碼使用 Java 1.0/1.1 容器，甚至新的程式碼有時候也會使用那些 classes。所以雖然撰寫新程式時你不該再使用舊容器，但你還是得了解它們才行。舊容器的功能極爲有限，並沒有太多需要說明的地方。它們已經是過去式了，所以我盡力克制自己不要去強調一些設計上令人厭惡的決定。

Vector 和 Enumeration

在 Java 1.0/1.1 中，**Vector** 是唯一可以自行擴增容量的序列，所以它被大量運用。但是其缺點多到難以於此詳列（請參考本書第一版 – 附於本書光

碟，也可自 *www.BruceEckel.com* 免費下載）。基本上你可以把它想像是個「函式名稱又長又不好用」的 **ArrayList**。在 Java 2 容器庫中，**Vector** 已經過調整，符合 **Collection** 規格和 **List** 規格。因此以下例子可以成功使用 **Collection2.fill()**。然而這其實是走上了一條不正確的道路，因爲這可能會誤導某些人以爲 **Vector** 變得更好了。其實它被 Java 2 含括進來只不過是爲了支援老舊的（Java 2 以前的）程式碼罷了。

Java 1.0/1.1 版的迭代器取名爲 "enumeration"（列舉），而不使用先前大家早已熟悉的術語。**Enumeration** interface 比 **Iterator** 小，只有兩個函式，名稱頗長。一個是 **boolean hashMoreElements()**，它會在「容器內部尚有元素」時回傳 **true**，另一個是 **Object nextElement()**，它會在「容器內部尚有元素」時回傳下一個元素（如果沒有元素了，就擲出異常）。

Enumeration 只是個 interface（介面），而不是個 implementation（實作物）。甚至即使新程式庫有時候也會運用舊的 **Enumeration** — 這令人十分遺憾，但通常不會造成傷害。雖說能夠的話最好使用 **Iterator**，但你也得有心理準備，程式庫可能會回傳給你一個 **Enumeration**。

此外，你可以透過 **Collections.enumeration()** 爲任意一個 **Collection** 產生一個 **Enumeration**，一如下例所示：

```
//: c09:Enumerations.java
// Java 1.0/1.1 Vector and Enumeration.
import java.util.*;
import com.bruceeckel.util.*;

class Enumerations {
  public static void main(String[] args) {
    Vector v = new Vector();
    Collections2.fill(
      v, Collections2.countries, 100);
    Enumeration e = v.elements();
    while(e.hasMoreElements())
      System.out.println(e.nextElement());
    // Produce an Enumeration from a Collection:
    e = Collections.enumeration(new ArrayList());
```

```
  }
} ///:~
```

Java 1.0/1.1 的 **Vector** 只有個 **addElement()**，但 **fill()** 會使用 **add()**，那是 **Vector** 被轉爲 **List** 時所帶入的。你可以呼叫 **elements()** 產生一個 **Enumeration**，然後便可用它來執行前向迭代（forward iteration）。

程式的最後一行產生一個 **ArrayList**，並使用 **enumeration()** 將 **ArrayList Iterator** 轉換成一個 **Enumeration**。因此如果你的舊程式需要 **Enumeration**，你還是可以使用新式容器。

Hashtable

一如你在本章效能評比相關篇幅中所見，**Hashtable** 和 **HashMap** 極爲相似，甚至連函式名稱都相像。因此在新程式中，沒有理由使用 **Hashtable** 來取代 **HashMap**。

Stack

stack 的觀念先前已經和 **LinkedList** 一起介紹過了。Java 1.0/1.1 的奇怪表現是，竟然不以 **Vector** 做爲 **Stack** 的基底，反而讓 **Stack** 繼承 **Vector**。因此 **Stack** 擁有 **Vector** 的所有特性和行爲，以及 **Stack** 自己的額外行爲。實在很難猜測這究竟是設計者明確認知下的有用作法，或只是一個幼稚的設計。

以下是 **Stack** 的簡單運用實例，將 **String** array 的每一行都推入 stack：

```
//: c09:Stacks.java
// Demonstration of Stack Class.
import java.util.*;

public class Stacks {
  static String[] months =
    "January", "February", "March", "April",
    "May", "June", "July", "August", "September",
    "October", "November", "December" };
  public static void main(String[] args) {
```

```
    Stack stk = new Stack();
    for(int i = 0; i < months.length; i++)
      stk.push(months[i] + " ");
    System.out.println("stk = " + stk);
    // Treating a stack as a Vector:
    stk.addElement("The last line");
    System.out.println(
      "element 5 = " + stk.elementAt(5));
    System.out.println("popping elements:");
    while(!stk.empty())
      System.out.println(stk.pop());
  }
} ///:~
```

months array 中的每一行都會經由 **push()** 被置於 **Stack** 內，並於稍後以 **pop()** 將它們自 stack 頂端取出。爲了特別強調，我在 **Stack** 物件身上執行 **Vector** 的動作，這沒問題，因爲，稍早所說的繼承關係使得「**Stack** 是一個 **Vector**」，因此所有可執行於 **Vector** 身上的操作，都可以執行於 **Stack** 身上，例如 **elementAt()**。

一如先前所說，當你想要 stack 的行爲時，你應該使用 **LinkedList**。

BitSet

如果想要高效率地儲存大量「開/關」資訊，**BitSet** 是很好的選擇。不過它的效率僅僅表現於空間。如果你需要高效率的存取時間，**BitSet** 會比原生的（語言支援的）array 稍慢一些。

此外，**BitSet** 的最小空間是 **long**：64 bits。這意味如果你所儲存的內容比較小，例如 8 bits，那麼 **BitSet** 就會浪費一些空間。因此如果空間對你是個問題，你最好撰寫自己的 class 或直接採用 array 來儲存開關旗標（on-off flags）。

一個正常的容器會隨著元素的加入而擴增其大小，**BitSet** 也是。以下示範 **BitSet** 的運作方式：

```
//: c09:Bits.java
// Demonstration of BitSet.
import java.util.*;
```

```
public class Bits {
  static void printBitSet(BitSet b) {
    System.out.println("bits: " + b);
    String bbits = new String();
    for(int j = 0; j < b.size() ; j++)
      bbits += (b.get(j) ? "1" : "0");
    System.out.println("bit pattern: " + bbits);
  }
  public static void main(String[] args) {
    Random rand = new Random();
    // Take the LSB of nextInt():
    byte bt = (byte)rand.nextInt();
    BitSet bb = new BitSet();
    for(int i = 7; i >=0; i--)
      if(((1 << i) &  bt) != 0)
        bb.set(i);
      else
        bb.clear(i);
    System.out.println("byte value: " + bt);
    printBitSet(bb);

    short st = (short)rand.nextInt();
    BitSet bs = new BitSet();
    for(int i = 15; i >=0; i--)
      if(((1 << i) &  st) != 0)
        bs.set(i);
      else
        bs.clear(i);
    System.out.println("short value: " + st);
    printBitSet(bs);

    int it = rand.nextInt();
    BitSet bi = new BitSet();
    for(int i = 31; i >=0; i--)
      if(((1 << i) &  it) != 0)
        bi.set(i);
      else
        bi.clear(i);
    System.out.println("int value: " + it);
    printBitSet(bi);
```

```java
    // Test bitsets >= 64 bits:
    BitSet b127 = new BitSet();
    b127.set(127);
    System.out.println("set bit 127: " + b127);
    BitSet b255 = new BitSet(65);
    b255.set(255);
    System.out.println("set bit 255: " + b255);
    BitSet b1023 = new BitSet(512);
    b1023.set(1023);
    b1023.set(1024);
    System.out.println("set bit 1023: " + b1023);
  }
} ///:~
```

這個程式使用亂數產生器來產生隨機的 **byte**、**short**、**int**，每一個都被轉換爲對應於 **BitSet** 的 bit 表示式。這麼做完全沒有問題，因爲 **BitSet** 是 64 bits，所以不會有任何一個數造成 **BitSet** 擴增大小。隨後程式產生了一個 512 bits 的 **BitSet**。建構式會配置比 bits 數更大一倍的空間，不過你也可以將編號 #1024（或更大）的 bit 設爲 true（譯註：意指 **BitSet** 會自動擴展）。

摘要

讓我們檢閱 Java 標準程式庫提供的各種容器：

1. array「將數值索引關聯至物件」。它持有型別已知的物件，所以搜尋物件時並不需要對搜尋結果做轉型動作。array 可以是多維度的，而且可以持有基本型別（primitives types）。不過 array 的容量在誕生之後便無法改變。

2. **Collection** 持有的是單一元素，**Map** 則持有相關聯的成對元素。

3. 和 array 一樣，**List** 也將數值索引關聯至物件。你可以將 array 和 **Lists** 想像成一種有序的容器（ordered container）。當你加入元素，**List** 會自動調整大小。但是 **List** 只能持有 **Object** reference，

無法持有基本型別（primitives types）。而且每當你自容器取出 **Object** reference，都得對結果進行轉型，才能拿來使用。

4. 如果你需要大量隨機存取，請使用 **ArrayList**。如果你會在 list 的中心處進行大量的安插或移除動作，請使用 **LinkedList**。

5. queues、deques、stacks 的行爲乃是透過 **LinkedList** 提供。

6. **Map** 是一種將物件（而非數值索引）關聯至其他物件的機制。**HashMap** 的設計思考著重於速度，**TreeMap** 則著重於如何讓 *key* 保持排序狀態，因此不像 **HashMap** 那麼快。

7. **Set** 不接受重複元素。**HashSet**s 提供最快的搜尋速度，**TreeSet**s 則使元素保持排序狀態。

8. 沒有必要在新程式中使用老舊的 **Vector**、**Hashtable**、**Stack**。

容器是你每天都會用到的工具。它可以讓你的程式更簡單、更具威力、也更有效率。

練習

某些經過挑選的題目，其解答置於《*The Thinking in Java Annotated Solution Guide*》電子文件中。僅需小額費用便可自 *www.BruceEckel.com*.網站取得。

1. 產生一個 **double** array，並充填元素（使用 **fill()** 並搭配 **RandDoubleGenerator**）。請印出結果。

2. 撰寫一個 **Gerbil**（沙鼠）class，令它具備 **int gerbilNumber**，並在建構式中加以初始化（和本章的 **Mouse** 範例類似）。讓它擁有一個 **hop()**，其功能是印出自己的編號以及「自己正在跳躍（hopping）」的訊息。請產生一個 **ArrayList** 並將多個 **Gerbil** 物件加於其內，然後使用 **get()** 走訪這個 **List**，並呼叫每個 **Gerbil** 的 **hop()**。

3. 修改練習 2，改使用 **Iterator** 來走訪 **List**（當呼叫 **hop()** 時）。

4. 使用練習 2 的 **Gerbil** class，並將它改置於 **Map** 內。將 **Gerbil** 的名稱當做 **String**（並視爲 *key*），令它關聯至你置於表格內的 **Gerbil**（視爲 *value*）。取得 **keySet()** 的 **Iterator**，並利用它來走訪 **Map**、搜尋每個 *key* 所對應的 **Gerbil**，以及印出 *key*、讓每個 **Gerbil** 執行 **hop()**。

5. 產生一個 **List**（請嘗試使用 **ArrayList** 和 **LinkedList**），並利用 **Collections2.countries** 充填內容。對此 list 排序並列印，然後將 **Collections.shuffle()** 反覆套用於此 list 身上，每次都列印內容，藉以觀察 **shuffle()** 每次是怎樣地將 list 內容打亂。

6. 證明除了 **Mouse** 之外，你無法將任何東西加入 **MouseList** 內。

7. 修改 **MouseList.java**，使它繼承 **ArrayList** 而非使用複合（composition）手法。闡述這個作法的問題所在。

8. 修正 **CatsAndDogs.java** 的問題：撰寫一個 **Cats** 容器（使用 **ArrayList**），令它只能接受、取出 **Cat** 物件。

9. 撰寫一個容器，將 **String** array 封裝於內，並且只能安插或取出 **Strings**，因此使用時沒有轉型的問題。如果安插元素時內部 array 的空間不足，這個容器應該自動調整大小。面對你這個容器和用以儲存 **Strings** 的 **ArrayList** 容器，請在 **main()** 之中比較兩者的效能。

10. 重複練習 9，但將容器改成儲存 **int**，並比較你的容器和儲存 **Integer** 物件之 **ArrayList** 的效能 — 包括對容器內的每個物件執行累加（incrementing）動作。

11. 使用 **com.bruceeckel.util** 的公用程式，產生各種 *primitives array* 和 **String** array，然後以適合的產生器（generator）充填每個 array，再以適當的 **print()** 將每個 array 內容印出。

12. 撰寫一個產生器（generator），使它產生你最喜歡的電影中的各個角色名稱（你可以選「白雪公主」或「星際大戰」），並在用完所有名稱時從頭再來一遍。使用 **com.bruceeckel.util** 中的公用程式將這些角色名稱分別填入 array、**ArrayList**、**LinkedList**、以及兩種 **Set** 內。然後印出每個容器的內容。

13. 撰寫一個 class，內含兩個 **String** 物件，並實作 **Comparable** 使得比較動作只與第一個 **String** 有關。使用 **geography** 產生器，將你這個 class 的物件充填至 array 和 **ArrayList** 內。請展示排序運作無誤。接下來，製作一個 **Comparator**，令它在比較動作中只在乎第二個 **String**，並請展示排序運作無誤。同時，請使用你的這個 **Comparator** 執行二分搜尋（binary search）。

14. 修改練習 13，依字母次序進行排序。

15. 使用 **Arrays2.RandStringGenerator** 來充填 **TreeSet**，並使用字母次序來排序。印出 **TreeSet** 的內容以檢驗排序是否正確。

16. 產生一個 **ArrayList** 和一個 **LinkedList**，並使用自動產生器 **Collection2.capitals** 來充填內容。使用一般的 **Iterator** 印出這兩個容器的內容，然後使用 **ListIterator** 將其中一個 list 安插到另一個 list 中，安插在任何你希望的位置。現在，在第一個 list 的尾端執行安插動作，並且往回（backward）移動。

17. 撰寫一個函式，使用 **Iterator** 來走訪 **Collection** 並印出容器內每個物件的 **hashCode()**。將物件填入各種不同型別的 **Collection**s 內，並將你的函式套用在每個容器身上。

18. 修正 **InfiniteRecursion.java** 中的問題。

19. 撰寫一個 class，然後以其物件爲初值產生一個 array。再根據這個 array 充填一個 **List**。使用 **subList()** 產生上述 **List** 的一個子集，並使用 **removeAll()** 將該子集自 **List** 中移除。

20. 修改第 7 章的練習 6，改用 **ArrayList** 來持有 **Rodent**s，並改用 **Iterator** 來走訪 **Rodent**s 序列。請千萬記得，**ArrayList** 僅能持有 **Object**s，所以在存取個別 **Rodent**s 時，你得搭配轉型動作。

21. 模仿 **Queue.java** 範例，撰寫一個 **Deque** class 並測試之。

22. 在 **Statistics.java** 中使用 **TreeMap**，並將「測試 **HashMap** 和 **TreeMap** 效能差異」的程式碼加入該程式。

23. 產生一個 **Map** 和一個 **Set**，內含所有以 'A' 爲首的國家名稱。

24. 使用 **Collections2.countries**，將同一份資料多次填入 **Set** 內，然後檢查 **Set** 面對重複資料是否只儲存一份。請在兩種 **Set** 上進行嘗試。

25. 以 **Statistics.java** 做爲開始，撰寫一個程式，可反覆執行測試，並檢查是否有哪個數字的出現頻率高於其他數字。

26. 使用 **Count** 物件的 **HashSet** 來重寫 **Statistics.java**（你得修改 **Counter** 俾使它能夠用於 **HashSet**）。哪一種作法看起來比較好呢？

27. 修改練習 13 中的 class，使它能夠搭配 **HashSet**s 使用，也可以做爲 **HashMap**s 的 *key*。

28. 模仿 **SlowMap.java**，撰寫一個 **SlowSet**。

29. 將 **Map1.java** 中的測試動作套用於 **SlowMap** 身上，檢查它是否運作正確。如果 **SlowMap** 的運作不正確，請加以修正。

30. 爲 **SlowMap** 實作出 **Map** interface 的剩餘部份。

31. 修改 **MapPerformance.java** 以含括 **SlowMap** 中的測試。

32. 修改 **SlowMap** 的內容，使它不再使用兩個 **ArrayList**s，而是持有唯一一個以 **MPair** 物件組成的 **ArrayList**。驗證修改過的版本是否正確運作。使用 **MapPerformance.java** 來測試新 **Map** 的速度。現在，修改 **put()**，使它在每一筆成對資料置入後進行

sort()。修改 **get()**，改用 **Collections.binarySearch()** 來搜尋 *key*。比較新舊版本的效能。

33. 爲 **CountedString** 新增一筆 **char** 資料，並同樣於建構式中加以初始化。修改 **hashCode()** 和 **equals()**，使它們將這個 **char** 納入考量。

34. 修改 **SimpleHashMap**，使它得以記錄碰撞的發生。將相同資料重複加入兩次，於是你便可以看到碰撞發生。

35. 修改 **SimpleHashMap**，使它得以記錄碰撞發生時所謂的「probes」個數。這個數字的意思是，當我們走訪 **LinkedList**，得在 **Iterator**s 上呼叫多少次 **next()** 才能找到想要搜尋的元素。

36. 實作 **SimpleHashMap** 的 **clear()** 和 **remove()**。

37. 爲 **SimpleHashMap** 實作 **Map** interface 的剩餘部份。

38. 將 **private rehash()** 加至 **SimpleHashMap** 中，並在負載因子超過 0.75 時呼叫它。一旦決定 rehashing，將 buckets 個數乘以 2，然後找出最鄰近而大於該値的質數，做爲新的 buckets 個數。

39. 模仿 **SimpleHashMap.java**，撰寫 **SimpleHashSet** 並加測試。

40. 修改 **SimpleHashMap**，以 **ArrayLists** 取代 **LinkedList**。修改 **MapPerformance.java** 以比較上述兩者效能。

41. 使用 JDK 的 HTML 文件（可自 *java.sun.com* 下載），找出 **HashMap** class。現在，產生一個 **HashMap**，充填元素，決定負載參數。測試此一 **Map** 的搜尋速度。然後產生一個新的 **HashMap**，給予較大的初始容量，並將舊 **Map** 複製過去。以此方式來增加速度。請在新 **Map** 身上再次執行搜尋測試。

42. 找出第 8 章的 **GreenhouseControls.java** 範例，其中含有三個檔案。在 **Controller.java** 中，class **EventSet** 不過是個容器，請改以 **LinkedList** 取代 **EventSet**。這不僅僅只是將 **EventSet** 換成 **LinkedList** 就好了，你還需要使用 **Iterator** 來走訪所有事件。

43. 挑戰題：撰寫自己的 hashed map class，使它專門爲某個特定的 *key* 型別（例如 **String**）量身打造。不要繼承 **Map**，而是重複其函式，讓 **put()** 和 **get()** 都只接受 **String** 物件（而非 **Object**s）做爲 *key*。任何與 *key* 相關的東西都不應該再使用泛式型別，而應該改用 **String**s，以避免向上轉型和向下轉型帶來的成本。你的目標是寫出最快產品。請修改 **MapPerformace.java**，用以測試你這份實作品和 **HashMap** 兩者之間的效能比較。

44. 挑戰題：在 Java 源碼庫中找出 **List** 原始碼。將其程式碼複製一份，並撰寫一個名爲 **intList** 的特殊版本。**intList** 僅持有 **int**s。思考一下，如果我們要爲每一個基本型別都製作一份特殊版本的 **List**，需要做些什麼事？如果你要製作一個 linked list class，有能力容納所有基本型別，又需要做些什麼事？如果 Java 提供參數化型別（*parameterized types*），便是爲此問題提供了一種自動化機制（當然還有其他好處）。

A: 物件的傳遞（Passing）和回傳（Returning）

如今，當你傳遞某個物件時，你應該已經相當接受一個事實：你傳遞的其實是個 reference。

在許多程式語言中，你可以使用該語言的一般方式來傳遞物件。大多數時候一切都會十分正常。但似乎總會有些時候你得進行一些不尋常的動作，事情突然間變得有點複雜（以 C++ 來說，變得極爲複雜）。Java 也不例外。重要的是，你必須了解傳遞物件並加以操作時，究竟會發生什麼樣的事。這份附錄將提供一些看法。

如果你先前曾經習慣於某種極具威力的程式語言，那麼，本附錄的另一個提問方式就是：「Java 有指標嗎？」。某些人宣稱，指標不僅難用，而且危險，因此是不好的東西；而由於 Java 是所有美德與光明之所在，並可卸下世俗的編程包袱，所以它不可能含有指標那種東西。然而比較精確的說法是，Java 有指標。是的，Java 之中除了基本型別，每個物件識別名稱都是指標。但它們的作用是受限的，不僅受編譯器的保護，也受執行期系統（runtime system）的保護。或者換句話說，Java 有指標，但沒有「指標運算」。這些正是我之前稱爲 "references" 的東西，你可以將它們想像爲「安全的指標」，就像小學裡的安全剪刀一樣。不過這種指標有時候效率不彰，而且令人感到厭煩。

reference 的傳遞

當你將 reference 傳入函式，reference 仍然會指向原先所指的同一物件。以下這個簡單實驗便說明了這一點：

```
//: appendixa:PassReferences.java
// Passing references around.

public class PassReferences {
  static void f(PassReferences h) {
    System.out.println("h inside f(): " + h);
  }
  public static void main(String[] args) {
    PassReferences p = new PassReferences();
    System.out.println("p inside main(): " + p);
    f(p);
  }
} ///:~
```

上述的列印指令中會自動呼叫 **toString()**，而 **PassReferences** 直接繼承自 **Object**，並未重新定義（覆寫）**toString()**。因此喚起的是 **Object** 的 **toString()**。此版本會印出物件的 class 名稱，以及物件（注意，不是 reference，是實際物件）所在的記憶體位址。輸出結果如下：

```
p inside main(): PassReferences@1653748
h inside f(): PassReferences@1653748
```

你可以看到，**p** 和 **h** 指向同一個物件。將引數傳入函式時，這種傳遞方式（譯註：*pass by reference*）的效率遠勝於複製一個新的物件（譯註：*pass by value*）。但是這個現象引領我們討論另一個重要議題。

別名（Aliasing）

別名（aliasing）代表的意思是，多個 references 被繫結至同一個物件，就像前例那樣。當某人對該物件進行塗寫動作時，別名就會引發問題。如果其他 reference 的擁有者並不預期該物件值會改變，他們便會對此結果感到意外。下面這個簡單例子可說明這一點：

```
//: appendixa:Alias1.java
// Aliasing two references to one object.

public class Alias1 {
  int i;
  Alias1(int ii) { i = ii; }
  public static void main(String[] args) {
    Alias1 x = new Alias1(7);
    Alias1 y = x; // Assign the reference
    System.out.println("x: " + x.i);
    System.out.println("y: " + y.i);
    System.out.println("Incrementing x");
    x.i++;
    System.out.println("x: " + x.i);
    System.out.println("y: " + y.i);
  }
} ///:~
```

以下這一行：

```
Alias1 y = x; // Assign the reference
```

產生了新的 **Alias1** reference，卻未被賦予一個以 **new** 產生的新物件，而是令它等同於一個原已存在的 reference。所以 reference **x** 的內容（也就是物件 **x** 的位址）被指派給 **y**。因此 **x** 和 **y** 都指向同一個物件。當 **x** 的 **i** 值在以下述句中遞增時：

```
x.i++;
```

y 的 **i** 值同受影響。從輸出結果可以觀察出這一點：

```
x: 7
y: 7
Incrementing x
x: 8
y: 8
```

一個好的解法就是：不要這麼做；是的，不要在同一個範圍（scope）中產生一個以上的 reference 別名。這會使你的程式碼更容易被理解和除錯。不過，當你將 reference 當做引數（這是 Java 的運作方式），便會自動產生

別名，因爲被產生出來的區域性的（local）reference 可以修改外界物件（此物件在 mothed 範圍之外被產生出來）。以下即是一例：

```
//: appendixa:Alias2.java
// Method calls implicitly alias their
// arguments.

public class Alias2 {
  int i;
  Alias2(int ii) { i = ii; }
  static void f(Alias2 reference) {
    reference.i++;
  }
  public static void main(String[] args) {
    Alias2 x = new Alias2(7);
    System.out.println("x: " + x.i);
    System.out.println("Calling f(x)");
    f(x);
    System.out.println("x: " + x.i);
  }
} ///:~
```

下面是輸出結果：

```
x: 7
Calling f(x)
x: 8
```

這個函式改變了引數內容，也就是外界物件的內容。當這種情況發生，你得決定這樣的結果對你是否有實質意義 — 這是你預期的行爲嗎？

一般來說，呼叫一個函式是爲了獲得其回傳值，或是爲了對呼叫者（某物件）的狀態進行改變（所謂函式就是：「發送訊息給某物件」的方法）。通常，呼叫函式很少是爲了處理其引數 — 該種行爲被稱爲「爲了副作用而呼叫函式」。因此當你撰寫「會修改引數內容」的函式時，必須清楚告訴你的使用者，並對該函式的運用以及它可能引發意外的行爲提出警告。由於這些令人困惑的特性及缺點，你最好避免更改引數值。

如果你需要在函式被呼叫期間修改引數，而你又不希望修改到外界物件，那麼你應該在你的函式中製作一份複本以保護外界物件。這正是本附錄的重點。

製作一個區域性副本（local copies）

讓我做個複習。Java 傳遞的引數皆以 reference 形式傳遞。也就是說，當你傳遞物件時，實際上傳遞的只是指向「存在於函式之外的某個物件」的 reference。所以如果你以該 reference 進行任何修改動作，都會修改到外界物件。此外：

- 引數傳遞過程中會自動產生別名（alias）。
- 沒有區域物件（local objects），只有區域性的（local）references。
- references 受範圍（scope）的限制，物件則否。
- 物件的壽命從來不是 Java 的討論議題（譯註：因爲有垃圾回收機制）。
- 沒有任何語言層次上的支援可以防止物件被修改。也就是說沒有任何語言層次上的支援可以防止別名（alias）造成的負面效應。

如果你只是要讀取物件中的資訊，而不是要加以修改，那麼 reference 是最有效率的一種引數傳遞方式。這很好，而且 Java 所賦予的 *pass by reference* 預設方式也是最有效率的。不過有時候你可能需要將物件視如「區域物件」一般地對待，這樣你所做的任何更動就只會影響區域內的這個副本，不會波及外界那個本尊。許多程式語言支援「在函式內自動產生外界物件副本」的能力[1]。Java 並不如此，但它允許你獲得這種效果。

---

[1]在 C 語言中，通常你處理極小的資料，所以預設採用 pass-by-value（傳值）。C++ 亦依循此種形式，但對物件而言 pass-by-value 通常不是最有效率的方式，此外在 C++ 中撰寫「支援 pass-by-value」的 classes，也是個令人頭痛的問題（譯註：我想作者指的是這個 class 必須有 copy constructor。雖然預設情況下編譯器會自動給你一個，但那只能做淺層拷貝，無法進行深層拷貝。另，C++也支援 pass by reference，唯其型式與 pass-by-value 略有差異）

Pass by value（傳值）

這帶來一個術語上的問題，而其爭論從來沒有停止過。這裡的術語是 *pass by value*（傳值），其意義視你對程式運作的理解程度而定。一般來說其意義是，你會取得你所傳入的物件的區域副本，但真正的問題在於你對「你所傳入的物件」是怎麼想的。當我們討論 *pass by value* 的意義時，有兩個涇渭分明的陣營：

1. Java 以 *by value*（傳值）方式來傳遞所有東西。當你傳遞基本型別的資料（進入函式），你會得到該資料的一個副本。當你傳遞 reference（進入函式），你會獲得該 reference 的一個副本。因此，所有東西其實都是以 *by value* 的方式傳遞。當然，這裡的前提是假設你知道你所傳遞的是 reference（而非物件本身）。但是 Java 的設計早已讓你忘記了你所處理的乃是 reference（而非物件本身）。它的設計希望你將 reference 視同物件，因爲當你呼叫函式時，它會自動對 reference 進行提領（*dereference*）動作。

2. 面對基本型別的資料，Java 是以值傳遞（這個毫無爭議），但面對物件，Java 是以址傳遞（*pass by reference*）。這是一種「將 reference 視爲物件別名」的世界觀，所以你並不會認爲你在傳遞 references，而會認爲你傳遞的是物件。由於將物件傳入函式時並不會獲得該物件的區域性副本，所以物件很明顯地並不是以值傳遞。在 Sun 公司內部，似乎有某些支持此種看法的力量，因爲有一個「被保留但未實作出」的關鍵字 **byvalue**。不過，沒有人知道這個關鍵字是否（或何時）會出現在世人面前。

將這兩個陣營的聲音都表達出來，並且宣稱「這和你如何看待 reference 有關」之後，我會試著規避這個問題。最後我要說，這其實不是那麼重要，重要的是你要知道 *pass by reference* 會讓呼叫者傳入的物件在非預期情況下被修改。

物件的克隆（Cloning objects）

譯註：clone 和 copy 的意義都是「複製」。但它們的深層意義並不相同。爲加以區分，我把 clone 譯爲克隆。克隆之於 clone，就像拷貝之於 copy 一樣，都是一種外來詞。雖然臺灣不流行說「克隆」，但是這個字眼在大陸是被普遍接受的 ☺。

製作某個物件的區域性副本，最有可能的原因是：你即將修改該物件，但不希望波及呼叫者手上的那一份本尊物件。如果你打算製作區域性副本，只要使用 **clone()** 即可，它就是基礎類別 **Object** 中定義為 **protected** 的那個 **clone()**，你得在「想要進行複製動作」的衍生類別中將它覆寫為 **public**。例如標準程式庫中的 class **ArrayList** 便覆寫了 **clone()**，所以我們可以呼叫 **ArrayList clone()** 如下：

```
//: appendixa:Cloning.java
// The clone() operation works for only a few
// items in the standard Java library.
import java.util.*;

class Int {
  private int i;
  public Int(int ii) { i = ii; }
  public void increment() { i++; }
  public String toString()
    return Integer.toString(i);
  }
}

public class Cloning {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++ )
      v.add(new Int(i));
    System.out.println("v: " + v);
    ArrayList v2 = (ArrayList)v.clone();
    // Increment all v2's elements:
    for(Iterator e = v2.iterator();
        e.hasNext(); )
      ((Int)e.next()).increment();
    // See if it changed v's elements:
    System.out.println("v: " + v);
  }
} ///:~
```

clone() 會回傳一個 **Object** 物件，你得將它轉型為適當型別。上述範例說明了 **ArrayList clone()** 並不會自動複製 **ArrayList** 所含的每一個物件，因此原先的 **ArrayList** 和複製後的 **ArrayList** 都指向同一堆物件（譯註：所謂一堆物件是指 list 中的那些元素）。這種作法常被稱為淺層拷貝

（*shallow copy*），因爲它只會複製物件的表面。實際的物件不但含有這個「表面」，還加上所有 references 所指向的物件（意指那些元素），以及所有那些元素所（可能）指向的物件…。這種關係通常被稱爲物件網絡（*web of objects*。譯註：見 p614）。整個龐雜的物件網絡的複製，就是所謂的深層拷貝（*deep copy*）。

你可以在輸出結果中觀察淺層拷貝效應。是的，對 **v2** 做的動作波及了 **v**：

```
v: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
v: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

不對 **ArrayList** 內的所有物件進行 **clone()**，或許是個合理的假設，因爲不能保證那些物件都「可被克隆」（cloneable）[2]。

賦予 class 克隆能力（cloneability）

即使 **clone()** 定義於「萬物之母」**Object** 中，但並不是每個 class 天生都具有克隆能力[3]。這似乎違反「base class 函式永遠可爲 derived classes 所用」的直覺概念。是的，Java 的克隆行爲便和上述直覺概念背道而馳；如果你希望某個 class 擁有克隆能力，你得特地加上一些程式碼，讓克隆機制起作用。

---

[2]這個字的拼法在字典中是找不著的。但是 Java 程式庫的確這麼用它。爲了減少混淆，我在這裡如法炮製。

[3]你好像可以針對這句話寫個反例，像這樣：

```
public class Cloneit implements Cloneable {
  public static void main (String[] args)
  throws CloneNotSupportedException {
    Cloneit a = new Cloneit();
    Cloneit b = (Cloneit)a.clone();
  }
}
```

不過，這是因爲 **main()** 是 **Cloneit** 的一個 method，因此擁有「呼叫 **protected** base class method」**clone()** 的權限，因而可以正常運作。如果你從另一個 class 呼叫之，是無法正確編譯的。

利用 protected 玩點把戲

爲了不讓你所寫的每個 class 預設都有克隆能力，**clone()** 在 base class **Object** 中被宣告爲 **protected**。這不僅意味單純使用（而非繼承）該 class 的用戶端程式在預設情況下無法使用它，也意味你無法透過 base class reference 來呼叫 **clone()**（雖然在某些情況下這可能滿實用的，例如以多型方式克隆許多 **Objects**）。這種方式其實是在編譯期讓你知道，你所處理的物件無法被複製。很奇怪的是， Java 標準程式庫的大多數 classes 都是不可克隆的。因此如果你這麼寫：

```
Integer x = new Integer(1);
x = x.clone();
```

會在編譯期收到錯誤訊息，告訴你 **clone()** 不可被取用（因爲 **Integer** 並未覆寫它，所以預設爲 **protected** 版本）。

不過，只要是在 **Object** derived class 中（噢，所有 classes 都如此），你便擁有呼叫 **Object.clone()** 的權限，因爲它被宣告爲 **protected** 而你是繼承者。那個 **clone()** 的功能十分有限 — 它會對 derived class object 進行位元逐一複製動作（*bitwise copy*）。因此它扮演著共通的複製行爲。你可以視需要將你的 **clone()** 宣告爲 **public**，使它可爲外界取用。至此，進行克隆時有兩個關鍵議題：

- 幾乎總是要呼叫 **super.clone()**。
- 將你的 **clone()** 宣告爲 **public**。

你或許會希望在更深層的 derived classes 中覆寫 **clone()**，否則你的（現被宣告爲 **public**）**clone()** 會被喚起而其動作可能不正確（儘管由於 **Object.clone()** 會進行實際的物件複製，導致還是可能有正確行爲）。**protected** 計倆只能運作一次，在你「第一次繼承不具克隆能力的 class 而又希望讓新 class 可被克隆」的時候。你的那個 class 的所有 derived classes 都可以使用你的 **clone()**，因爲 Java 無法在繼承過程中降低函式的存取權限。也就是說一旦某個 class 具備克隆能力，那麼它的所有 derived classes 都可以克隆，除非你使用某種機制關閉克隆能力（稍後介紹）。

Cloneable 介面實作

要讓某個物件得到完整的克隆能力，你得做一件事情：實作出 **Cloneable interface**。這個介面有點奇怪，因爲它是空的！

```
interface Cloneable {}
```

之所以要實作這個空的 **interface**，不是因爲你打算把你的 class 向上轉型爲 **Cloneable** 並呼叫其函式。在這裡，**interface** 的運用被某些人視爲一種 "hack"，因爲它代表了某種原先意圖以外的東西。**Cloneable interface** 的實作被視爲一種「融入 class 型別系統」的標記。

Cloneable interface 的存在有兩個理由。第一，你可能會取得一個「向上轉型至基礎型別」的 reference，而你不知道能否對它進行克隆。這種情況下你可以使用關鍵字 **instanceof**（第 12 章曾經介紹過），判斷某個 reference 是否指向可被克隆的物件：

```
if(myReference instanceof Cloneable) // ...
```

第二個原因是，克隆能力已被混入整個設計中，然而你也許不希望所有類型的物件都能夠被克隆。**Object.clone()** 會檢驗某個 class 是否實作出 **Cloneable**，如果沒有，便會擲出 **CloneNotSupportedExcpetion** 異常。所以一般而言，提供克隆能力的同時，你一定得實作 **Cloneable**。（譯註：也有人將這種「內容爲空，用來標示具有某種性質」的 interface 稱爲 *Marker Interface* 設計樣式。**Cloneable** 和 **java.io.Serializable** 皆屬此類）

成功的克隆

一旦了解 **clone()** 的實作細節後，你便可以撰寫可被克隆的 classes，因而得以爲它產生區域副本：

```
//: appendixa:LocalCopy.java
// Creating local copies with clone().
import java.util.*;

class MyObject implements Cloneable {
```

```
  int i;
  MyObject(int ii) { i = ii; }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("MyObject can't clone");
    }
    return o;
  }
  public String toString() {
    return Integer.toString(i);
  }
}

public class LocalCopy {
  static MyObject g(MyObject v) {
    // Passing a reference, modifies outside object:
    v.i++;
    return v;
  }
  static MyObject f(MyObject v) {
    v = (MyObject)v.clone(); // Local copy
    v.i++;
    return v;
  }
  public static void main(String[] args) {
    MyObject a = new MyObject(11);
    MyObject b = g(a);
    // Testing reference equivalence,
    // not object equivalence:
    if(a == b)
      System.out.println("a == b");
    else
      System.out.println("a != b");
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    MyObject c = new MyObject(47);
    MyObject d = f(c);
    if(c == d)
```

```
      System.out.println("c == d");
    else
      System.out.println("c != d");
    System.out.println("c = " + c);
    System.out.println("d = " + d);
  }
} ///:~
```

首先，**clone()** 必須是可取用的，所以你得將它宣告爲 **public**。其次，**clone()** 一開始處你應該呼叫 base class **clone()**；這裡所呼叫的 **clone()** 是 **Object** 所定義的那個版本。由於它被宣告爲 **protected**，所以 **Object** derived classes 可以取用，所以你可以呼叫它。

Object.clone() 會知道物件大小，爲它配置足夠的記憶體空間，並將舊物件的內容複製到新物件中。此即所謂「位元逐一複製（bitwise copy）」，這也是你通常意識並預期的 **clone()** 行爲。但是 **Object.clone()** 執行其動作之前必須先檢查 class 是否爲 **Cloneable**，也就是說它是否實作了 **Cloneable** interface。如果沒有，那麼 **Object.clone()** 便會擲出異常 **CloneNotSupportedException**，表示無法加以克隆。爲此，你應該將 **super.clone()** 呼叫動作擺到 **try-catch** 區塊中，以便捕捉應該永遠不會出現的異常（因爲你的確實作了 **Cloneable** interface）。

在 **LocalCopy** 中，**g()** 和 **f()** 說明了兩種引數傳遞方式的差異。**g()** 採用 *by reference* 傳遞方式，因此會修改外界物件並回傳指向該外界物件的 reference。**f()** 則對引數進行克隆，因而切斷了與原物件間的關係；接下來便可進一步做任何它想要做的動作，甚至將新物件的 reference 回傳，也不會影響原物件。請注意下面這行述句，看起來有點奇怪：

```
v = (MyObject)v.clone();
```

這正是產生區域副本之處。爲了避免你被這種述句迷惑，請千萬記得，這種頗爲奇怪的程式編寫手法在 Java 中非常適當，因爲每個物件識別名稱實際上都是 references。所以，reference **v** 被用來克隆（**clone()**）出其所指物件的複本，並回傳一個「指向基礎型別 **Object**」的 reference（**Object.clone()** 的確是這麼定義的），我們必須將該基礎型別再轉型爲適當型別。

在 **main()** 中執行上述兩個函式，測試兩種引數傳遞方式所引發的不同效應。輸出如下：

```
a == b
a = 12
b = 12
c != d
c = 47
d = 48
```

千萬注意，Java 的相等性（equivalence）測試並不檢查被比較的兩個物件內容是否相同。運算子 == 和 != 只比較 references。如果 references 所代表的位址相同，它們就是指向同一個物件，因而被視爲相等。所以==和 != 運算子所檢查的其實是「兩個 references 是否是同一個物件的別名（alias）」！

Object.clone() 的效應

當 **Object.clone()** 被呼叫，實際上究竟發生什麼動作，致使你在自己的 class 中覆寫 **clone()** 時一定得呼叫 **super.clone()** 呢？根類別（也就是 **Object**）中的 **clone()** 負責產生正確大小的儲存空間，並進行位元逐一複製動作，將原始物件的內容複製到新物件的儲存空間。也就是說它不僅僅只是製作出儲存空間並複製一個 **Object**，它實際上還會計算被複製物的精確大小，並加以複製。由於上述所有動作都由根類別所定義的 **clone()** 執行，所以你可以猜測這整個程序一定動用了 RTTI 機制，用來判斷被複製的實際物件。透過這種方式，**clone()** 便可以產出恰當大小的儲存空間，並對它進行正確的位元逐一複製動作（bitwise copy）。

不論你打算怎麼做，克隆程序的第一個動作正常來說應該就是呼叫 **super.clone()**，如此才能製作出一個一模一樣的複製品，做爲克隆行爲的基礎。而後你便可以執行其他必要動作，完成整個複製程序。

如果想確切知道所謂「其他的必要動作」究竟是什麼，你得確切知道 **Object.clone()** 爲你做了些什麼事。特別是，它是否自動克隆了物件內含的所有 references？下面這個例子主要便是用來檢驗此點：

```
//: appendixa:Snake.java
// Tests cloning to see if destination
// of references are also cloned.

public class Snake implements Cloneable {
  private Snake next;
  private char c;
  // Value of i == number of segments
  Snake(int i, char x) {
    c = x;
    if(--i > 0)
      next = new Snake(i, (char)(x + 1));
  }
  void increment() {
    c++;
    if(next != null)
      next.increment();
  }
  public String toString() {
    String s = ":" + c;
    if(next != null)
      s += next.toString();
    return s;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Snake can't clone");
    }
    return o;
  }
  public static void main(String[] args) {
    Snake s = new Snake(5, 'a');
    System.out.println("s = " + s);
    Snake s2 = (Snake)s.clone();
    System.out.println("s2 = " + s2);
    s.increment();
    System.out.println(
      "after s.increment, s2 = " + s2);
```

```
  }
} ///:~
```

Snake 由許多段組成，每一段的型別都是 **Snake**。也就是說它是個單鏈串列（singly linked list）。每一段都以遞迴方式產生：每遞迴產生一個 **Snake**，就將建構式的第一引數遞減 1，直到遞減至 0 爲止。爲了讓每一段都有一個獨一無二的識別標籤，其第二引數 **char** 會在每次遞迴建構時遞增 1。

increment() 會遞迴地遞增每個識別標籤值，於是你才能夠觀察其改變。**toString()** 會遞迴印出每個識別標籤。輸出結果爲：

```
s = :a:b:c:d:e
s2 = :a:b:c:d:e
after s.increment, s2 = :a:c:d:e:f
```

這表示 **Object.clone()** 只複製了第一段，也就是說它只做淺層拷貝。如果你希望複製整條蛇，也就是進行深層拷貝，你得在你所覆寫的 **clone()** 中執行額外動作。

通常你會在「具備克隆能力的 class」的某個 derived class 中呼叫 **super.clone()**，用以確保所有 base class 的行爲（含 **Object.clone()**）都會發生。只要對你的物件中的每個 reference 都呼叫其 **clone()**，便可達到這個目的；否則這些 references 只會成爲原物件的別名（alias）。這和呼叫建構式的方式是一樣的：先呼叫 base class 的建構式，然後是下一個衍生類別的建構式，依此類推，直到最後一層衍生類別。差別只在於 **clone()** 並非建構式，所以編譯器沒有爲它提供一個自動進行機制。你得設法自行完成。

克隆一個複合物件（composed object）

當你試著對複合物件進行深層拷貝時，會遭遇某個問題。你得假設成員物件的 **clone()** 會輪番地在其 references 上進行深層拷貝，而且不斷遞迴下去。這是一個承諾，它實際上代表，如果希望深層拷貝起作用，你得在所

有 classes 中控制所有程式碼，或至少得有足夠的知識，認識深層拷貝中的所有相關 classes，以便檢驗它們是否都會正確執行自己的深層拷貝。

下面這個範例說明，進行複合物件的深層拷貝時，你得完成哪些動作：

```
//: appendixa:DeepCopy.java
// Cloning a composed object.

class DepthReading implements Cloneable {
  private double depth;
  public DepthReading(double depth)
    this.depth = depth;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      e.printStackTrace(System.err);
    }
    return o;
  }
}

class TemperatureReading implements Cloneable {
  private long time;
  private double temperature;
  public TemperatureReading(double temperature) {
    time = System.currentTimeMillis();
    this.temperature = temperature;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      e.printStackTrace(System.err);
    }
    return o;
  }
}
```

```
class OceanReading implements Cloneable {
  private DepthReading depth;
  private TemperatureReading temperature;
  public OceanReading(double tdata, double ddata){
    temperature = new TemperatureReading(tdata);
    depth = new DepthReading(ddata);
  }
  public Object clone() {
    OceanReading o = null;
    try {
      o = (OceanReading)super.clone();
    } catch(CloneNotSupportedException e) {
      e.printStackTrace(System.err);
    }
    // Must clone references:
    o.depth = (DepthReading)o.depth.clone();
    o.temperature =
      (TemperatureReading)o.temperature.clone();
    return o; // Upcasts back to Object
  }
}

public class DeepCopy {
  public static void main(String[] args) {
    OceanReading reading =
      new OceanReading(33.9, 100.5);
    // Now clone it:
    OceanReading r =
      (OceanReading)reading.clone();
  }
} ///:~
```

DepthReading 和 **TemperatureReader** 極類似；兩者都只含基礎類型資料。因此 **clone()** 十分簡單：呼叫 **super.clone()** 並回傳結果。請注意這兩個 classes 的 **clone()** 一模一樣。

OceanReading 由 **DepthReading** 物件和 **TemperatureReader** 物件組成。爲了進行深層拷貝，其 **clone()** 必須克隆「**OceanReading** 內的所有 references」。爲了完成這個任務，**super.clone()** 的傳回結果必須

被轉型爲 **OceanReading**（這樣你才能夠取用 references 的 **depth** 和 **temperature**）。

對 ArrayList 進行深層拷貝

讓我們回頭看看本附錄先前的 **ArrayList** 實例。這一次我讓 **Int2** class 變成爲「可克隆」（cloneable），於是可對 **ArrayList** 進行深層拷貝：

```
//: appendixa:AddingClone.java
// You must go through a few gyrations
// to add cloning to your own class.
import java.util.*;

class Int2 implements Cloneable {
  private int i;
  public Int2(int ii) { i = ii; }
  public void increment() { i++; }
  public String toString() {
    return Integer.toString(i);
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Int2 can't clone");
    }
    return o;
  }
}

// Once it's cloneable, inheritance
// doesn't remove cloneability:
class Int3 extends Int2 {
  private int j; // Automatically duplicated
  public Int3(int i) { super(i); }
}

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

```
    Int2 x = new Int2(10);
    Int2 x2 = (Int2)x.clone();
    x2.increment();
    System.out.println(
      "x = " + x + ", x2 = " + x2);
    // Anything inherited is also cloneable:
    Int3 x3 = new Int3(7);
    x3 = (Int3)x3.clone();

    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++ )
      v.add(new Int2(i));
    System.out.println("v: " + v);
    ArrayList v2 = (ArrayList)v.clone();
    // Now clone each element:
    for(int i = 0; i < v.size(); i++)
      v2.set(i, ((Int2)v2.get(i)).clone());
    // Increment all v2's elements:
    for(Iterator e = v2.iterator();
        e.hasNext(); )
      ((Int2)e.next()).increment();
    // See if it changed v's elements:
    System.out.println("v: " + v);
    System.out.println("v2: " + v2);
  }
} ///:~
```

Int3 繼承自 **Int2**，而且加入了新的基礎型成員 **int j**。你可能會以為你得再覆寫 **clone()** 以確保 **j** 會被複製，但事實並非如此。當 **Int2 clone()** 因 **Int3 clone()** 而被喚起，它會呼叫 **Object.clone()**，而 **Object.clone()** 會判斷它所處理的是 **Int3**，於是複製 **Int3** 物件中的所有位元。只要你沒有加入任何需要被克隆的 references，呼叫 **Object.clone()** 一次，就可以完成所有必要複製 — 不論 **clone()** 定義點位於階層體系中多深的位置。

你可以看到，對 **ArrayList** 進行深層拷貝需要哪些動作：在 **ArrayList** 被克隆後，你得逐一走訪 **ArrayList** 所指的每一個物件，並逐一克隆。如果你對 **HashMap** 進行深層拷貝，也需要類似的舉動。

這個例子的其他部份用來顯示克隆結果。一旦物件被克隆，你便可以改變它而不波及原物件。

透過 serialization 進行深層拷貝

當你思考 Java object serialization（第 11 介紹過）時，你可能會觀察到，某個物件被次第寫出（serialized）而後再被反向讀回（deserialized），事實上就是被克隆了一份。

那麼，爲什麼不利用 serialization 執行深層拷貝呢？以下這個例子透過執行時間的計算，對兩種作法進行比較：

```
//: appendixa:Compete.java
import java.io.*;

class Thing1 implements Serializable {}
class Thing2 implements Serializable {
  Thing1 o1 = new Thing1();    // 譯註：複合
}

class Thing3 implements Cloneable {
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Thing3 can't clone");
    }
    return o;
  }
}

class Thing4 implements Cloneable {
  Thing3 o3 = new Thing3();    // 譯註：複合
  public Object clone() {
    Thing4 o = null;
    try {
      o = (Thing4)super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Thing4 can't clone");
```

```
    }
    // Clone the field, too:
    o.o3 = (Thing3)o3.clone();
    return o;
  }
}

public class Compete {
  static final int SIZE = 5000;
  public static void main(String[] args)
  throws Exception {
    Thing2[] a = new Thing2[SIZE];
    for(int i = 0; i < a.length; i++)
      a[i] = new Thing2();
    Thing4[] b = new Thing4[SIZE];
    for(int i = 0; i < b.length; i++)
      b[i] = new Thing4();
    long t1 = System.currentTimeMillis();
    ByteArrayOutputStream buf =
      new ByteArrayOutputStream();
    ObjectOutputStream o =
      new ObjectOutputStream(buf);
    for(int i = 0; i < a.length; i++)
      o.writeObject(a[i]);
    // Now get copies:
    ObjectInputStream in =
      new ObjectInputStream(
        new ByteArrayInputStream(
          buf.toByteArray()));
    Thing2[] c = new Thing2[SIZE];
    for(int i = 0; i < c.length; i++)
      c[i] = (Thing2)in.readObject();
    long t2 = System.currentTimeMillis();
    System.out.println(
      "Duplication via serialization: " +
      (t2 - t1) + " Milliseconds");
    // Now try cloning:
    t1 = System.currentTimeMillis();
    Thing4[] d = new Thing4[SIZE];
    for(int i = 0; i < d.length; i++)
      d[i] = (Thing4)b[i].clone();
```

```
    t2 = System.currentTimeMillis();
    System.out.println(
      "Duplication via cloning: " +
      (t2 - t1) + " Milliseconds");
  }
} ///:~
```

Thing2 和 **Thing4** 都含有成員物件（譯註：也就是說他們都是複合物件），所以會進行某種深層拷貝動作。有趣的是，雖然很容易產生 **Serializable** classes，但複製它們時卻需要更多勞動。而 cloning（複製）雖然一開始需要花費許多力氣設定 class，實際的物件複製行動卻簡單得多。程式執行結果正說明了這一點。以下是程式執行三次的各自結果：

```
Duplication via serialization: 940 Milliseconds
Duplication via cloning: 50 Milliseconds

Duplication via serialization: 710 Milliseconds
Duplication via cloning: 60 Milliseconds

Duplication via serialization: 770 Milliseconds
Duplication via cloning: 50 Milliseconds
```

Serialization（次第讀寫）和 cloning（克隆）之間有著極大的效率差異。此外你也應該注意，serialization 所花費的時間似乎變動極大，cloning 則較爲穩定。

將克隆能力加到繼承體系的更下層

當你撰寫新的 class 並衍生自 **Object**，**Object** 在預設情況下不具克隆能力（下一節便會看到）。如果你不自行爲你的 class 賦予克隆能力，它便沒有這份能力。不過你可以在繼承體系的任一層加入克隆能力，那麼該層之下的每一層就都變成可克隆了，像這樣：

```
//: appendixa:HorrorFlick.java
// You can insert Cloneability
// at any level of inheritance.
import java.util.*;

class Person {}
```

```
class Hero extends Person {}
class Scientist extends Person
    implements Cloneable {
  public Object clone() {
    try {
      return super.clone();
    } catch(CloneNotSupportedException e) {
      // this should never happen:
      // It's Cloneable already!
      throw new InternalError();
    }
  }
}
class MadScientist extends Scientist {}

public class HorrorFlick {
  public static void main(String[] args) {
    Person p = new Person();
    Hero h = new Hero();
    Scientist s = new Scientist();
    MadScientist m = new MadScientist();

    // p = (Person)p.clone(); // Compile error
    // h = (Hero)h.clone(); // Compile error
    s = (Scientist)s.clone();
    m = (MadScientist)m.clone();
  }
} ///:~
```

加入克隆能力之前，編譯器會阻止你的克隆行為。克隆能力被加至 **Scientist** 後，**Scientist** 及其衍生類別便都帶有克隆能力。

為什麼要有這種奇怪的設計？

上述這些看起來像個奇怪的體制，是嗎？是的，它本來就怪。你可能會懷疑，為什麼要以這種方式進行呢？這種設計背後的意涵又是什麼呢？

最早的 Java 是為了用來控制硬體設備，當時絕對沒有考量到 Internet。在類似 Java 這樣的一般性程式語言中，讓程式員能夠克隆物件似乎是很合理

的。因此 **clone()** 被放在根類別 **Object** 中，但當時它被宣告爲 **public**，所以你無論怎樣都可以克隆任一個物件。這似乎是最有彈性的方式，但它最終損害了什麼呢？

唔，當 Java 被視爲終極的 Internet 程式語言時，局勢有所改變。突然間 Java 得考慮安全問題，這些問題和物件的使用有關：你不會希望每個人都有能力克隆你希望保密的物件吧？所以你看到了，有許多補救措施套用在原先簡單直覺的架構上：**clone()** 如今在 **Object** 中成了 **protected**、你得覆寫 **clone()** 並實作 **Cloneable**、然後你得處理異常。

只有當你呼叫 **Object clone()** 時，**Cloneable** interface 才有用處。因爲 **Object clone()** 會於執行期檢查你的 class 是否實作出 **Cloneable**。不過，基於一致性考量（而且 **Cloneable** 不過就是個空的 interface），你無論如何還是應該實作它。

克隆能力（cloneability）的控制

你可能會聯想到，如果要移除克隆能力，只需將 **clone()** 宣告爲 **private** 就好了。但這其實行不通，因爲你無法在 derived class 中降低 base class 函式的存取權限。所以事實並非你想像的那麼簡單。人們終究需要控制某個物件的克隆能力。現實中你可以採取許多不同的態度：

1. 漠不關心。不進行任何克隆相關動作，這意味你的 class 無法被克隆（cloned），但其 derived classes 可以加入克隆能力（如果它想要的話）。只有當預設的 **Object.clone()** 能夠合理處理你的 class 內的所有資料欄位時，這一點才適用。

2. 支援 **clone()**。實作 **Cloneable** 並覆寫 **clone()**。覆寫 **clone()** 時應該呼叫 **super.clone()** 並捕捉所有異常（那麼你覆寫的 **clone()** 就不會擲出任何異常）。

3. 有條件地支援克隆（cloning）。如果你的 class（例如各種容器）持有的 references 指向一些可能（或可能不）具有克隆能力的物件，那麼你的 **clone()** 應該試著克隆每一個「你所持有的 references」的所指物件，並在它們擲出異常時直接將異常丟回給程式員。舉個例子，有一種特別的 **ArrayList** 能夠克隆它所持有的所有物件，當你撰寫此類 **ArrayList** 時，你不知道用戶端程式員可能會把什麼樣類型的物件放進來，所以你不知道它們是否可被克隆。

4. 不實作 **Cloneable**，但以 **protected** 方式覆寫 **clone()**，使所有欄位都能被正確複製。透過這種方式，所有繼承自你的這個 class 的人都可以覆寫 **clone()**，並且呼叫 **super.clone()** 獲得正確的拷貝行為。請注意，你的 **clone()** 可以（並應該）呼叫 **super.clone()**，即使 **super.clone()** 預期的是個 **Cloneable** 物件（如果不是這樣，它會擲出異常）。沒有人能夠直接於你所撰寫的 class 物件中呼叫 **clone()**，只能夠透過 derived class（如果它想正確運作，得實作 **Cloneable** 才行）加以呼叫。

5. 嘗試阻止克隆動作：不實作 **Cloneable**，並覆寫 **clone()** 使它擲出異常。你的這個 class 的 derived classes 都必須在它們自己的 **clone()** 中呼叫 **super.clone()**，這個方法才行得通。否則程式員還是可以規避這個方法。

6. 將你的 class 宣告為 **final**，防止克隆動作發生。如果 **clone()** 並未被你的這個 class 的父類別（或祖先類別）覆寫，那就沒辦法這麼做。如果它在上層類別中被覆寫了，那麼請再次覆寫，並宣告有可能擲出 **CloneNotSupportedException**。將這個 class 宣告為 **final** 是唯一可以保證「克隆動作不會發生」的作法。此外，在處理保密性物件或必須控制產出物件的個數時，你應該將所有建構式都宣告為 **private**，並提供一個（或以上）產出物件的函式。透過這種方式，這些函式就可以限制產出的物件個數，以及它們的產出條件。（你可以從 *www.BruceEckel.com* 下載《*Thinking in Patterns with Java*》，其中介紹了一個例子：*singleton* 設計樣式）

下例示範各種方法，用來實作克隆（cloning）機能，或關閉繼承體系下層的克隆能力：

```
//: appendixa:CheckCloneable.java
// Checking to see if a reference can be cloned.

// Can't clone this because it doesn't
// override clone():
class Ordinary {}

// Overrides clone, but doesn't implement
// Cloneable:
class WrongClone extends Ordinary {
  public Object clone()
      throws CloneNotSupportedException {
    return super.clone(); // Throws exception
  }
}

// Does all the right things for cloning:
class IsCloneable extends Ordinary
    implements Cloneable {
  public Object clone()
      throws CloneNotSupportedException {
    return super.clone();
  }
}

// Turn off cloning by throwing the exception:
class NoMore extends IsCloneable {
  public Object clone()
      throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
  }
}

class TryMore extends NoMore {
  public Object clone()
      throws CloneNotSupportedException {
    // Calls NoMore.clone(), throws exception:
    return super.clone();
```

```
  }
}

class BackOn extends NoMore {
  private BackOn duplicate(BackOn b) {
    // Somehow make a copy of b
    // and return that copy. This is a dummy
    // copy, just to make the point:
    return new BackOn();
  }
  public Object clone() {
    // Doesn't call NoMore.clone():
    return duplicate(this);
  }
}

// Can't inherit from this, so can't override
// the clone method like in BackOn:
final class ReallyNoMore extends NoMore {}

public class CheckCloneable {
  static Ordinary tryToClone(Ordinary ord) {
    String id = ord.getClass().getName();
    Ordinary x = null;
    if(ord instanceof Cloneable) {
      try {
        System.out.println("Attempting " + id);
        x = (Ordinary)((IsCloneable)ord).clone();
        System.out.println("Cloned " + id);
      } catch(CloneNotSupportedException e) {
        System.err.println("Could not clone "+id);
      }
    }
    return x;
  }
  public static void main(String[] args) {
    // Upcasting:
    Ordinary[] ord =
      new IsCloneable(),
      new WrongClone(),
      new NoMore(),
```

```
      new TryMore(),
      new BackOn(),
      new ReallyNoMore(),
    };
    Ordinary x = new Ordinary();
    // This won't compile, since clone() is
    // protected in Object:
    //! x = (Ordinary)x.clone();
    // tryToClone() checks first to see if
    // a class implements Cloneable:
    for(int i = 0; i < ord.length; i++)
      tryToClone(ord[i]);
  }
} ///:~
```

Ordinary 代表我們在本書中常看到的 classes 類型：不支援但也不阻止克隆。不過如果你取得一個 **Ordinary** object reference，而它被向上轉型時，你便無法知道它是否可被克隆。

WrongClone 示範錯誤的克隆機制（cloning）實作方式。它覆寫 **Object.clone()**，並將之宣告爲 **public**，卻未實作出 **Cloneable**，所以當 **super.clone()** 被呼叫時（引發對 **Object.clone()** 的呼叫），便會擲出 **CloneNotSupportedException**，導致無法進行克隆。

在 **IsCloneable** 中，你可以看到它的克隆動作完全正確：覆寫了 **clone()** 並實作出 **Cloneable**。不過這個 **clone()** 和本例稍後的其他函式都沒有捕捉 **CloneNotSupportedException**，而是將它傳給呼叫者，因此呼叫者必須在 **try-catch** 區段中包覆它。通常你應該在自己的 **clone()** 中捕捉 **CloneNotSupportedException**，而不是將它傳出。本例這種作法純粹是爲了教育意義。

Class **NoMore** 試圖採用 Java 設計者所希望的方式關閉克隆能力：在 derived class **clone()** 中擲出 **CloneNotSupportedException**。**TryMore clone()** 會適當呼叫 **super.clone()**（本例將是 **NoMore.clone()**），於是擲出異常，阻止克隆發生。

如果程式員在覆寫的 **clone()** 中不呼叫 **super.clone()**，會發生什麼事呢？你可以在 **BackOn** 中看到可能發生的事情。這個 class 會使用另一個函式 — **duplicate()** — 來製作目前物件的副本：在 **clone()** 中呼叫 **duplicate()** 而不呼叫 **super.clone()**。異常永遠不會被擲出，而新類別擁有克隆能力。你無法藉由擲出異常的方式來阻止他人製作出擁有克隆能力的 class。唯一能夠達此目的的方法顯示於 **ReallyNoMore** 身上。這個 class 被宣告爲 **final**，因此無法被繼承。這意味如果 **clone()** 於 **final** class 中擲出異常，它將無法因繼承而被衍生類別修改（覆寫），因而保證無法克隆。要知道，你無法在繼承體系的任意一層呼叫 **Object.clone()**，你只能呼叫 **super.clone()**。因此如果你的物件涉及安全性，你應該會希望將這些物件宣告爲 **final**。

你在 **CheckCloneable** 中看到的第一個函式是 **tryToClone()**。它接受任何 **Ordinary** 物件，並以 **instanceof** 檢查此物件是否可以克隆。如果是，便將該物件轉型爲 **IsCloneable**，然後呼叫其 **clone()** 並將結果轉回 **Ordinary**，捕捉所有被擲出的異常。請注意，這裡使用執行期辨識機制（請參考第 12 章）印出 class 名稱。

main() 會產生不同類型的 **Ordinary** 物件，並將它們向上轉型放入 **Ordinary** 陣列。之後的前兩行程式碼會產生出一個平常的 **Ordinary** 物件並嘗試克隆它。不過這段程式碼無法正確編譯，因爲 **clone()** 是 **Object** 的 **protected** 函式。程式碼的其餘部份會依序走訪陣列中的元素，並試著克隆每個物件，最後逐一回報每個物件究竟是克隆成功或失敗。輸出結果爲：

```
Attempting IsCloneable
Cloned IsCloneable
Attempting NoMore
Could not clone NoMore
Attempting TryMore
Could not clone TryMore
Attempting BackOn
Cloned BackOn
Attempting ReallyNoMore
Could not clone ReallyNoMore
```

總的來說，如果你希望某個 class 可被克隆，你應該：

1. 實作 **Cloneable** interface。
2. 覆寫 **clone()**。
3. 於你的 **clone()** 中呼叫 **super.clone()**。
4. 於你的 **clone()** 中捕捉異常。

這樣可以得到最合宜的結果。

copy 建構式

克隆機制（cloning）的建立似乎是個過於複雜的程序，似乎應該有其他替代方案。你可能會想到一個方法（尤其如果你曾經是 C++ 程式員），那便是撰寫一種特殊的建構式，其職責便是進行物件的複製。在 C++中，這種建構式被稱爲 *copy* 建構式。這看起來似乎是個明顯的解法，事實上這個方法行不通。以下即爲一例：

```
//: appendixa:CopyConstructor.java
// A constructor for copying an object of the same
// type, as an attempt to create a local copy.

class FruitQualities {
  private int weight;
  private int color;
  private int firmness;
  private int ripeness;
  private int smell;
  // etc.
  FruitQualities() { // Default constructor
    // do something meaningful...
  }
  // Other constructors:
  // ...
  // Copy constructor:
  FruitQualities(FruitQualities f) {
    weight = f.weight;
    color = f.color;
```

```
    firmness = f.firmness;
    ripeness = f.ripeness;
    smell = f.smell;
    // etc.
  }
}

class Seed {
  // Members...
  Seed() { /* Default constructor */ }
  Seed(Seed s) { /* Copy constructor */ }
}

class Fruit {
  private FruitQualities fq;
  private int seeds;
  private Seed[] s;
  Fruit(FruitQualities q, int seedCount)
    fq = q;
    seeds = seedCount;
    s = new Seed[seeds];
    for(int i = 0; i < seeds; i++)
      s[i] = new Seed();
  }
  // Other constructors:
  // ...
  // Copy constructor:
  Fruit(Fruit f) {
    fq = new FruitQualities(f.fq);
    seeds = f.seeds;
    // Call all Seed copy-constructors:
    for(int i = 0; i < seeds; i++)
      s[i] = new Seed(f.s[i]);
    // Other copy-construction activities...
  }
  // To allow derived constructors (or other
  // methods) to put in different qualities:
  protected void addQualities(FruitQualities q) {
    fq = q;
  }
  protected FruitQualities getQualities() {
```

```
    return fq;
  }
}

class Tomato extends Fruit {
  Tomato() {
    super(new FruitQualities(), 100);
  }
  Tomato(Tomato t) { // Copy-constructor
    super(t); // Upcast for base copy-constructor
    // Other copy-construction activities...
  }
}

class ZebraQualities extends FruitQualities {
  private int stripedness;
  ZebraQualities() { // Default constructor
    // do something meaningful...
  }
  ZebraQualities(ZebraQualities z) {
    super(z);
    stripedness = z.stripedness;
  }
}

class GreenZebra extends Tomato {
  GreenZebra() {
    addQualities(new ZebraQualities());
  }
  GreenZebra(GreenZebra g) {
    super(g); // Calls Tomato(Tomato)
    // Restore the right qualities:
    addQualities(new ZebraQualities());
  }
  void evaluate() {
    ZebraQualities zq =
      (ZebraQualities)getQualities();
    // Do something with the qualities
    // ...
  }
}
```

```
public class CopyConstructor {
  public static void ripen(Tomato t) {
    // Use the "copy constructor":
    t = new Tomato(t);
    System.out.println("In ripen, t is a " +
      t.getClass().getName());
  }
  public static void slice(Fruit f) {
    f = new Fruit(f); // Hmmm... will this work?
    System.out.println("In slice, f is a " +
      f.getClass().getName());
  }
  public static void main(String[] args) {
    Tomato tomato = new Tomato();
    ripen(tomato); // OK
    slice(tomato); // OOPS!
    GreenZebra g = new GreenZebra();
    ripen(g); // OOPS!
    slice(g); // OOPS!
    g.evaluate();
  }
} ///:~
```

乍看之下這似乎有點奇怪。當然，水果擁有一些特性，可是爲什麼不直接將代表這些特性的資料成員放到 **Fruit** class 中呢？有兩個可能原因。第一個原因是，你可能希望將來能夠很輕易地新增或改變這些特性。請注意，**Fruit** 擁有 **protected addQualities()** 俾使 derived class 得以那麼做。（也許你會認爲合理的作法是讓 **Fruit** 擁有一個 **protected** 建構式並在其中接受一個 **FruitQualities** 引數。但因爲建構式不會被繼承，所以在第二層或更下層的 classes 中無法運用）。將水果特性擺到另一個 class 中，你就有了比較大的彈性，甚至可以在某個 **Fruit** 物件存活期間改變這些特性。

將 **FruitQualities** 設計爲獨立物件的第二個原因是，如果你想加入新特性，或是想透過繼承或多型來改變其行爲，你就會想要這麼做。請注意，對 **GreenZebra** 來說（這是一種番茄品種，我已栽培好一陣子，現在長得極好），其建構式會呼叫 **addQualities()** 並將 **ZebraQualities** 物件傳

入，後者繼承 **FruitQualities**，可被附於 base class 的 **FruitQualities** reference 中。當然，當 **GreenZebra** 運用 **FruitQualities** 時，它得將 **FruitQualities** 向下轉型至正確型別（就像 **evaulate()** 那樣），不過它總是會知道，正確的型別是 **ZebraQualties**。

你會看到程式裡有個 **Seed** class，**Fruit**（按定義將帶著它自己的種子）[4] 則含有一個 **Seed** 陣列。

最後，請注意每個 class 都擁有 *copy* 建構式，而每個 *copy* 建構式都必須負責呼叫 base class 和成員物件的 *copy* 建構式，以進行深層拷貝。class **CopyConstructor** 測試了 *copy* 建構式：由 **ripen()** 接收一個 **Tomato** 引數，並爲它完成一個拷貝建構（copy construction），複製該物件：

```
t = new Tomato(t);
```

slice() 接收更一般化的 **Fruit** 物件，並且也加以複製：

```
f = new Fruit(f);
```

main() 以各種不同類型的 **Fruit** 對此進行測試。以下爲其輸出結果：

```
In ripen, t is a Tomato
In slice, f is a Fruit
In ripen, t is a Tomato
In slice, f is a Fruit
```

這就曝露出問題來了。當 **slice()** 內的拷貝建構動作發生在 **Tomato** 身上時，其結果不再是個 **Tomato** 而僅僅只是個 **Fruit**。它失去了番茄的特性。此外，當你傳入的是 **GreenZebra** 時，**ripen()** 和 **slice()** 會分別將它轉換成 **Tomato** 和 **Fruit**。因此很不幸地，對我們來說，當我們試著在 Java 裡頭製作出某個物件的區域性副本時，拷貝建構法並不適用。

[4] 除了可憐的鱷梨（avocado）— 此種水果已經被改良成完完全全的「肥厚」。

爲什麼適用於 C++ 卻不適用於 Java？

copy 建構式是 C++ 的核心部份，因爲它會自動產出物件的區域性副本。然而上例已經證明此法在 Java 行不通。爲什麼呢？在 Java 裡頭，我們操作的通通是 reference，而在 C++ 中你可以擁有 reference，也可以直接傳遞物件。當你以 *by value* 方式傳遞物件，該物件就會被複製，這正是 C++ *copy* 建構式的重要用途之一。這種方式在 C++ 裡頭一點問題也沒有。你應該記住，這種方式在 Java 裡頭行不通，所以不要用。

唯讀類別（Read-only classes）

雖然 **clone()** 產生的區域性副本在適當情況下得到了我們想要的結果，但這卻是「強迫程式員（函式開發者）必須擔起責任以免除別名（alias）帶來負面效應」的一個例子。如果你正在開發某個程式庫，屬於通用性質且將廣爲人所使用，你無法假設你的物件永遠都能夠在適當的地點被克隆，這種情況下會發生什麼事呢？或者，更可能的情況是，如果爲了效率考量你希望允許別名現象（*aliasing*）存在，以避免非必要的物件副本，但你又不希望沾染別名引發的副作用時，又當如何？

有一個方法，那便是產生所謂「恆常物件（*immutable objects*）」，此種物件隸屬於唯讀類別（*read-only classes*）。你可以在 class 子句中不定義任何「會更動物件內部狀態」的函式。這麼一來即使有別名現象（aliasing）也不會造成影響，因爲你只能讀取其內部狀態。所以即使程式中有許多地方都對同一個物件進行讀取，也不會構成問題。

舉一個「恆常物件」的例子：Java 標準程式庫中所有基本型別的「包覆類別（*wrapper classes*）便是。你可能已經察覺到這一點：如果想將 **int** 儲存於類似 **ArrayList** 的容器中（它只接收 **Object** references），你可以爲 **int** 包覆一個標準程式庫提供的 **Integer** class：

```
//: appendixa:ImmutableInteger.java
// The Integer class cannot be changed.
import java.util.*;

public class ImmutableInteger {
```

```
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++)
      v.add(new Integer(i));
    // But how do you change the int
    // inside the Integer?
  }
} ///:~
```

Integer class（以及所有基本型別的包覆類別）都以一種簡單方式來實作此種恆常性質：它們不具任何「可讓你修改物件內容」的函式。

如果你的確需要一個物件，不但持有某個基本型別，又可被修改，那麼你就得自己撰寫。幸好這做起來很簡單：

```
//: appendixa:MutableInteger.java
// A changeable wrapper class.
import java.util.*;

class IntValue
  int n;
  IntValue(int x) { n = x; }
  public String toString()
    return Integer.toString(n);
  }
}

public class MutableInteger {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++)
      v.add(new IntValue(i));
    System.out.println(v);
    for(int i = 0; i < v.size(); i++)
      ((IntValue)v.get(i)).n++;
    System.out.println(v);
  }
} ///:~
```

如果「**IntVvalue** 預設初值爲零」（那就不需要建構式）能被接受，而且不在意其值是否能被印出（那就不需要 **toString()**），那麼它甚至可以更簡單些：

```
class IntValue { int n; }
```

取出元素並將之轉型使用，會造成一點點不便，但那是 **ArrayList**（而非 **IntValue**）的事情。

撰寫一個唯讀類別（read-only classes）

你可以撰寫自己的唯讀類別，以下即爲一例：

```
//: appendixa:Immutable1.java
// Objects that cannot be modified
// are immune to aliasing.

public class Immutable1 {
  private int data;
  public Immutable1(int initVal) {
    data = initVal;
  }
  public int read() { return data; }
  public boolean nonzero() { return data != 0; }
  public Immutable1 quadruple() {
    return new Immutable1(data * 4);
  }
  static void f(Immutable1 i1) {
    Immutable1 quad = i1.quadruple();
    System.out.println("i1 = " + i1.read());
    System.out.println("quad = " + quad.read());
  }
  public static void main(String[] args) {
    Immutable1 x = new Immutable1(47);
    System.out.println("x = " + x.read());
    f(x);
    System.out.println("x = " + x.read());
  }
} ///:~
```

所有資料都被宣告爲 **private**，而且你看不到任何可修改資料的 **public** 函式。實際上會修改物件的函式只有 **quadruple()**，但它會產出新的 **Immutable1** 物件，不影響原先那一個。

f() 接收 **Immutable1** 物件，並於其上執行各種動作。**main()** 的輸出說明 **x** 值並不會被改變。因此我們可對 **x** 物件進行多次別名動作（aliasing）而不會造成傷害。因爲 **Immutable1** class 的設計可保護物件不受改變。

恆常性（immutability）的缺點

恆常類別一開始似乎可以提供一種優雅解法。但是當你需要修改該型別的物件時，你得忍受因產生新物件而造成的額外負擔，以及可能更頻繁的垃圾回收動作。對某些 classes 來說這不是問題。但是對另一些 classes 而言（例如 **String**），這種代價過於昂貴。

解決方法便是撰寫另一個扮演副手的 class，它是可修改的。當你需要大量的修改動作時，你可以切換到那個可修改的 class 上，修改結束後再切換爲不可修改的（恆常的）class。

現在我改變上一個例子，示範新的寫法：

```
//: appendixa:Immutable2.java
// A companion class for making
// changes to immutable objects.

class Mutable {
  private int data;
  public Mutable(int initVal) {
    data = initVal;
  }
  public Mutable add(int x)
    data += x;
    return this;
  }
  public Mutable multiply(int x) {
    data *= x;
    return this;
  }
```

```
  public Immutable2 makeImmutable2() {
    return new Immutable2(data);
  }
}

public class Immutable2 {
  private int data;
  public Immutable2(int initVal) {
    data = initVal;
  }
  public int read() { return data; }
  public boolean nonzero() { return data != 0; }
  public Immutable2 add(int x)
    return new Immutable2(data + x);
  }
  public Immutable2 multiply(int x) {
    return new Immutable2(data * x);
  }
  public Mutable makeMutable() {
    return new Mutable(data);
  }
  public static Immutable2 modify1(Immutable2 y){
    Immutable2 val = y.add(12);
    val = val.multiply(3);
    val = val.add(11);
    val = val.multiply(2);
    return val;
  }
  // This produces the same result:
  public static Immutable2 modify2(Immutable2 y){
    Mutable m = y.makeMutable();
    m.add(12).multiply(3).add(11).multiply(2);
    return m.makeImmutable2();
  }
  public static void main(String[] args) {
    Immutable2 i2 = new Immutable2(47);
    Immutable2 r1 = modify1(i2);
    Immutable2 r2 = modify2(i2);
    System.out.println("i2 = " + i2.read());
    System.out.println("r1 = " + r1.read());
    System.out.println("r2 = " + r2.read());
```

```
  }
} ///:~
```

Immutable2 內含的一些函式如 **add()** 和 **multiply()**，一如前述，當需要修改物件時，便會先產生一個物件副本再修改該副本，用以維持物件的恆常性。扮演副手角色的 class 名爲 **Mutable**，也擁有 **add()** 和 **multiply()**，但它們會對 **Mutable** 物件進行修改，不會製造出副本。此外，**Mutable** 有個函式會運用其資料產生一個 **Immutable2** 物件，反之亦然。

modify1() 和 **modify2()** 這兩個 static 函式雖然作法不同但結果相同。在 **modify1()** 中所有動作皆於 **Immutable2** class 完成，而且你可以發現過程中總共產生四個新的 **Immutable2** 物件；每當 **val** 被重新賦值，前一個物件就成爲垃圾。

在 **modify2()** 中你可以看到，第一個動作是依據 **Immutable2 y** 產出 **Mutable**（這和先前所見呼叫 **clone()** 是一樣的，但這次產生的是不同型別的物件）。接著使用 **Mutable** 物件來執行許多修改動作，而不需要產出許多物件。最後再將該物件轉換回 **Immutable2**。這個過程產生了兩個物件（**Mutable** 和最後的 **Immutable2**）而不是四個。

當以下情況發生，上述方法十分有用：

1. 你需要恆常物件（immutable objects），而且...
2. 你常需要進行大量修改，或是...
3. 產生新的恆常物件需要付出昂貴代價。

恆常不變的 Strings

考慮下列程式碼：

```
//: appendixa:Stringer.java

public class Stringer {
  static String upcase(String s) {
    return s.toUpperCase();
```

```
  }
  public static void main(String[] args) {
    String q = new String("howdy");
    System.out.println(q); // howdy
    String qq = upcase(q);
    System.out.println(qq); // HOWDY
    System.out.println(q); // howdy
  }
} ///:~
```

當 **q** 被傳入 **upcase()** 時，它實際上是 **q** reference 的一個副本。這個 reference 所指物件停置於某實際位置上（譯註：物件本身並沒有產生副本）。references 被傳遞時會被複製。

觀察 **upcase()**，你會看到，傳入的 reference 名爲 **s**，而且 **s** 只存活於 **upcase()** 本體。當 **upcase()** 執行完畢，區域性的 **s** 便被摧毀。**upcase()** 會傳回結果，也就是原先字串內容全被轉爲大寫後的結果。當然，傳回的是 reference 而不是物件本身。事實上被傳回的那個 reference 指向新物件，原先的 **q** 值並未被改變。這樣的動作是怎麼發生的呢？

不明顯的常數（Implicit constants）

如果你這麼寫：

```
String s = "asdf";
String x = Stringer.upcase(s);
```

你會希望 **upcase()** 修改其引數值嗎？一般來說答案是否定的，因爲對程式碼讀者來說，通常引數被視爲「提供給函式的一些資訊」，而不是「即將被修改的某個事物」。這是一個很重要的保證，讓程式碼更易於撰寫和理解。

在 C++ 中，這種保證重要到必須加入一個特殊關鍵字 **const**，讓程式員確保 reference（也就是 C++ 的 pointer 或 reference）無法被用來修改原物件。但這麼一來 C++ 程式員就得多花一些力氣並且絕不能忘記在各個必要地點運用 **const**。這很容易讓人混淆並且遺忘。

重載後的 '+'，以及 StringBuffer

透過前述技巧，**String** 物件被設計成具有恆常性。如果你檢視 **String** class 的線上說明文件（本章稍後有一些摘要），你會發現 class 之內任何一個函式，如果會修改 **String** 內容，實際上都會產生一個「含有修改結果」的 **String** 並傳回。原先的 **String** 不會被更動。Java 並沒有類似 C++ **const** 這樣的性質可讓編譯器支援物件恆常特性。如果你想要這樣的性質，你得自己處理一些類似 **String** 所做的動作。

由於 **String** 物件是恆常的（immutable），所以你可以對某個 **String** 產生任意個別名（alias）。由於它是唯讀的，所以某個 reference 不可能修改「會影響其他 references」的什麼東西。唯讀物件漂亮地解決了別名問題。

如果你有必要修改物件，藉由「產生一個物件，內含修改後的結果」這種方式，似乎可以解決。不過有些動作是很沒有效率的，例如將作用於 **String** 物件上的運算子 '+' 加以重載，便是一個例子。「重載」（overloading）意味當它被用於某個 class 時，有額外的意義。（針對 **String** 而完成的 '+' 和 '+='，是 Java 唯一能夠（被）重載的運算子。Java 不允許程式員重載其他運算子[5]。

面對 **String** 物件，'+' 讓你能夠將 **String**s 內容輕鬆串接在一塊：

```
String s = "abc" + foo + "def" + Integer.toString(47);
```

[5] C++ 允許程式員任意重載運算子。由於這往往是個複雜的過程（請參考《*Thinking in C++*，2nd *edition*》第 10 章，Prentice-Hall, 2000），所以 Java 設計者認爲這是個「不好」的功能，不應該放到 Java 裡頭。它其實沒有差勁到不適合擺到 Java 裡頭。而且諷刺得很，運算子重載在 Java 中應該比在 C++ 中輕鬆多了。從 Python（具有垃圾回收機制和簡單易行的運算子重載）中可以觀察到這一點。

你可以想像它可能的運作方式：**String** "abc" 可能有個 **append()**，它會產出新的 **String** 物件，後者含有 "abc" 並串接 **foo** 的內容。新的 **String** 物件接著再產生一個新的 **String** 物件，附加 "def "，依此類推。

這麼做當然行得通，但這種作法會產生許多 **String** 物件，最終卻只是爲了將它們組成新的 **String**。最後你會得到許多中間產物（**String** 物件），它們還得被垃圾回收機制收回去。我猜想 Java 設計者曾經嘗試過這種作法（軟體設計的第一課：除非你寫了程式親做嘗試，並讓某些機制動起來，否則永遠不會對系統有任何了解）。我也猜想 Java 設計者發現這種作法有著難以被接受的低效率。

解決辦法便是具恆常性的副手類別（mutable companion class），類似稍早前示範的例子。對 **String** 來說，**StringBuffer** 就是扮演其副手角色。編譯器會自動產出一個 **StringBuffer** 來評估某些算式，尤其當（重載版的）運算子 + 和 += 用於 **String** 物件時。下面這個例子展示發生的動作：

```
//: appendixa:ImmutableStrings.java
// Demonstrating StringBuffer.

public class ImmutableStrings {
  public static void main(String[] args) {
    String foo = "foo";
    String s = "abc" + foo +
      "def" + Integer.toString(47);
    System.out.println(s);
    // The "equivalent" using StringBuffer:
    StringBuffer sb =
      new StringBuffer("abc"); // Creates String!
    sb.append(foo);
    sb.append("def"); // Creates String!
    sb.append(Integer.toString(47));
    System.out.println(sb);
  }
} ///:~
```

當產生 **String s** 時，編譯器所做的事情，效果大約和之後使用 **sb** 所做的事情一樣：產出 **StringBuffer** 並使用 **append()** 將新字元直接加到

StringBuffer 物件中（而不再是每次都產出一份新物件）。雖然這種作法較有效率，但每次當你產生諸如 "abc" 或 "def" 之類以雙引數括住的字串時，編譯器會將它們轉換爲 **String** 物件。所以產出的物件個數，會超過你的預期。儘管如此 **StringBuffer** 還是能夠提供較好的效率。

String 和 StringBuffer

下面是一份簡單說明，對象是 **String** 和 **StringBuffer** 都擁有的函式。或許你能夠從中體會兩者的互動方式。這個表格並未涵蓋每一個函式，只涵蓋本討論主題中的重要函式。經過重載的函式會被摘要整理於同一列中。

首先是 **String** class：

函式名稱	**Arguments**（引數），**Overloading**（重載）	用途
Constructor	重載：*default* 建構式、**String**、**StringBuffer**、**char** 陣列、**byte** 陣列。	產生一個 **String** 物件
length( )		獲知 **String** 字元個數。
charAt()	**int** 索引	獲知 **String** 中某位置上的字元。
getChars( ), getBytes( )	來源端的起始和結束位置，目的端的陣列名稱，及其起始放置點（索引）。	將 **chars** 或 **bytes** 複製到外部陣列。
toCharArray( )		得到一個 **char [ ]**，內含 **String** 所有字元
equals( ), equals-IgnoreCase( )	一個用於比較的 **String**。	進行兩個 **Strings** 的相等性測試。
compareTo( )	一個用於比較的 **String**。	將本身的 **String** 和傳入的引數兩者依字典排列

函式名稱	**Arguments**（引數），**Overloading**（重載）	用途
		次序（lexicographical order）比較結果，傳回負值、零、或正值。注意，大小寫有別！
regionMatches( )	本身這個 **String** 的偏移位置、另一個 **String**（引數）及其偏移位置和比較長度。重載版本提供了一個「不分大小寫」的選項	傳回一個 **boolean**，代表指定的段落是否內容相符。
startsWith( )	做爲測試標準（是否爲起首）的 **String**。重載版本提供一個用於引數身上的起始偏移位置。	傳回一個 **boolean**，代表本身字串是否以引數內容爲起首。
endsWith( )	做爲測試標準（是否爲結尾）的 **String**。	傳回一個 **boolean**，代表本身字串是否以引數內容爲結尾。
indexOf( ), lastIndexOf( )	重載：**char**、**char** 和起始索引、**String**、**String** 和起始索引。	如果傳入的引數不爲 **String** 所涵蓋，傳回-1。反之則傳回該引數在 **String** 中的起始位置。**lastIndexOf()** 從末端反向搜尋。
substring( )	重載：起始索引、起始索引和終止索引。	傳回一個新的 **String**，內含指定的字元(s)。
concat( )	一個 **String**，將被用來做爲串接內容。	傳回一個新的 **String**，內含原始字串內容和新增字串內容。
replace( )	欲被取代的舊字元、欲取代的新字元。	傳回一個取代後的新 **String**。如果條件不

函式名稱	**Arguments**（引數），**Overloading**（重載）	用途
		符，則傳回舊 **String**。
toLowerCase( ) **toUpperCase( )**		傳回大小寫更換後的新 **String**。如果不需變動，傳回舊 **String**。
trim( )		傳回移除前後空白字元後的新 **String**。如果不需變動，傳回舊 **String**
valueOf( )	重載：**Object**、**char []**、**char[]** 和偏移和長度，**boolean**、**char**、**int**、**long**、**float**、**double**。	傳回一個 **String**，內含引數的「字元表述」（character representation）
intern( )		產生唯一一個 **String**，代表每個獨一無二的字元序列。

你會發現，每個 **String** 函式在它們改變字串內容時，都小心翼翼地傳回一個新的 **String** 物件。也請注意，如果字串內容無需更動，那麼該函式便會直接傳回一個 reference 指向原本的 **String**。這種作法可以節省儲存空間以及非必要的額外負擔。

以下是 **StringBuffer** class：

函式名稱	**Arguments**（引數），**overloading**（重載）	用途
Constructor	重載：*default* 建構式, 欲產生的緩衝區長度，**String** 的來源。	產生一個新的 **StringBuffer** 物件。

toString( )		根據 **StringBuffer** 產生一個 **String** 物件。
length( )		**StringBuffer** 內的字元個數。
capacity( )		傳回目前配置的空間大小。
ensure-Capacity( )	一個整數，指出需求容量。	要求 **StringBuffer** 持有至少某數量的空間。
setLength( )	一個整數，用來表示緩衝區內的字元字串的新長度。	截短或擴展原本的字元字串。如果是擴展，新增空間全部填 nulls。
charAt( )	一個整數，用來表示某個元素位置。	傳回緩衝區某位置上的 **char**。
setCharAt( )	一個整數，用來表示某個元素位置；一個新的 **char**，用來放置於該位置上。	修改某位置上的值。
getChars( )	來源端的起始和結束位置，目的端的陣列名稱，及其起始放置點（索引）。	將 **chars** 複製到一個外部陣列中。注意，並不像 **String** 那樣有個 **getBytes( )**。
append( )	重載：**Object, String, char[], char[]**，偏移位置和長度；另一重載版本：**boolean, char, int, long, float, double.**	引數將被轉換爲一個字串，並附加於目前緩衝區的尾端。如果必要，緩衝區會變大。
insert( )	重載，每一個版本都有第一引數，用來指示安插起始位置：**Object, String, char[], boolean, char, int, long, float, double.**	第二引數被轉換爲一個字串，並安插於目前緩衝區中的安插起始位置上。如果必要，緩衝區會變大。
reverse( )		將緩衝區內的字元全數逆轉位置。

最常被使用的函式是 **append()**。當編譯器評估含有 "+" 和 "+=" 的 **String** 運算式時，便會使用它。**insert()** 的形式與其類似。兩個函式都會在緩衝區中進行大量修改而不產生出新物件。

Strings 是特殊的東西

現在，你已經看到了，**String** class 不只是 Java 中的一個普通 class。**String** 存在許多特殊情況，不僅僅只是一個 Java 內建型別。事實上由雙引號括起的字串都會被編譯器和特殊的重載運算子 "+" 和 "+=" 轉換爲 **String**。這份附錄還告訴了你其他一些特殊情況：**String** 透過提供輔助功能的 **StringBuffer()** 十分小心地維護恆常性（immutability）。

摘要

由於 Java 的一切事物都是 reference，也由於每個物件都被生成於 heap 之中，當它不再被使用時，會被垃圾回收機制收回，所以改變了物件操作方式，尤其是在傳遞和回傳物件時。在 C 或 C++ 裡頭，如果你想在函式內初始化某一塊儲存空間，你或許會要求使用者將該儲存空間的位址傳入，否則你就必須爲「究竟誰會移除那塊儲存空間」傷腦筋。因此這種函式的介面和可理解度就比較複雜。在 Java 中你完全不需要爲了「儲存空間的移除」或「需要某個物件時它是否存在」等問題而煩惱，因爲 Java 已經自動爲你處理掉了。你可以在需要物件的時候產生它，並從那個時候開始永遠都不需要考量物件傳遞責任上的技巧。是的，只要傳遞 reference 就好了。有時候這種方式帶來的簡化不值一哂，但有時候它的影響力十分驚人。

所有這些神奇力量，衍生出兩個缺點：

1. 由於額外的記憶體管理動作而付出效率上的代價（雖然可能極輕微）。而且當程式執行時，總會摻雜些許的不確定性（因為當可用記憶體不足時，垃圾回收器便會強行介入）。不過對大多數應用程式而言，好處大過於缺點。程式中特別著重效率的部份，可使用原生函式（native methods，請參考附錄 B）來撰寫。

2. 別名（aliasing）：有時候在非刻意情況下，兩個 references 會指向同一個物件。只有當兩個 references 被假設指向同一個明確物件時，這才會成為問題。這正是你需要多加注意的地方。必要時請對物件執行 **clone()**，以避免另一個 reference 對於非預期的改變感到驚訝。你也可以撰寫「恆常物件」來提供別名上的效率。此種恆常物件的所有操作函式都會傳回一個同型（或不同型）物件，但無論如何不會更改原始物件的內容，因此，任何人即使對原始物件有個別名，也絕不會遇上物件內容被改變的情況。

某些人認為 Java 克隆機制是一種笨拙的設計，並對此感到厭煩，於是實作出自己的克隆機制[6]，並且完全不使用 **Object.clone()**。因而規免了實作 **Cloneable** 和捕捉 **CloneNotSupportedException** 的必要性。這當然是一種合理的方法，而且由於 Java 標準程式庫很少支援 **clone()**，所以這種方法似乎也很安全。只要你不呼叫 **Object.clone()**，你就不需要實作 **Cloneable**，也不需要捕捉相關異常，因此這麼做似乎頗能被接受。

[6] Doug Lea（他協助我解決這個主題）對我提出了這樣的建議。他說他只不過是為每一個 class 產生一個名為 **duplicate( )** 的函式，就行了。

練習

某些經過挑選的題目，其解答置於《*The Thinking in Java Annotated Solution Guide*》電子文件中。僅需小額費用便可自 *www.BruceEckel.com*.網站取得。

1. 請說明別名（aliasing）的第二個層次。撰寫一個函式，令它接受一個 reference，但並不修改該 reference 所指物件的內容。不過這個函式會呼叫另一個函式並將該 reference 傳入，而第二個函式卻會修改 reference 所指物件。

2. 撰寫一個 **myString** class，其中包含一個 **String** 物件，可於建構式中透過引數來設定初值。加入 **toString()** 和 **concatenate()**。後者會將 **String** 物件附加於你的內部字串尾端。請爲 **myString** 實作 **clone()**。撰寫兩個 **static** 函式，令它們都接收 **myString** reference **x** 引數並呼叫 **x.concatenate("test")**。但第二個函式會先呼叫 **clone()**。請測試這兩個函式並展示其不同的結果。

3. 撰寫一個 **Battery** class，其中內含 **int** 值，代表電池的編碼（獨一無二的識別名稱）。將它宣告爲 **Cloneable** 並爲它撰寫一個 **toString()**。接著撰寫一個 **Toy** class，其內包含一個 **Battery** 陣列和一個 **toString()**，後者可印出所有 **Battery** 物件。爲 **Toy** 撰寫 **clone()**，由它自動克隆所有 **Battery** 物件。測試方法：克隆 **Toy** 並列印結果。

4. 修改 **CheckCloneable.java**，讓所有 **clone()** 都捕捉 **CloneNotSupportedException**，而不是直接將異常傳遞給呼叫者。

5. 運用搭配「可修改之副手類別」（mutable-companion-class）的技巧，撰寫含有 **int**、**double**、**char** 陣列的恆常類別（immutable class）。

6. 修改 **Compete.java**，將更多物件加到 **Thing2** 和 **Thing4** classes 中，並試著觀察時間和複雜度之間的變化 — 是否爲單純的線性關係？或是更複雜？

7. 從 **Snake.java** 著手，撰寫 **Snake** 的深層拷貝（deep copy）版本。

8. 繼承 **ArrayList**，並讓其 **clone()** 執行深層拷貝（deep copy）。

B: Java 原生介面

The Java Native Interface，JNI

這份附錄由 Andrea Provaglio（*www.AndreProvaglio.com*）撰寫，並在他的允許之下用於此處。

Java 程式語言及其標準 API 十分豐富，可堪用來撰寫成熟的應用程式。但是，在某些特殊情況中，你得呼叫 non-Java 程式碼。例如，如果你想存取作業系統特有的功能、想要和特殊的硬體設備溝通、想要重複使用過去撰寫的 non-Java 程式碼、或是想要實作極重視時間效率的程式碼，你就會想呼叫 non-Java 程式碼。

想要和 non-Java 程式碼銜接，編譯器和虛擬機器必須提供專門的支援才行。而且還得有額外的工具，將 Java 程式碼對應至 non-Java 程式碼。Non-Java 程式碼的呼叫標準，是由 JavaSoft 提供的所謂「Java 原生介面（Java Native Interface，JNI）」，也正是本附錄的介紹內容。這並不是一份非常深入的論述，甚至某些例子還會假設你已經具備小部分相關觀念和技術。

JNI 是個頗爲豐富的編程介面，它讓你得以於 Java 應用程式中呼叫原生函式（native methods）。JNI 於 Java 1.1 被加入，和 Java 1.0 的對等機制 — 也就是「原生函式介面（Native Method Interface，NMI）」 — 維持了某種程度的相容性。NMI 的設計特質不適用於所有虛擬機制。基於這個原因，Java 語言的未來版本中可能不會再支援 NMI。本處亦不介紹 NMI。

目前 JNI 的設計只能用來和 C 或 C++ 所寫成的原生函式（native methods）相接。透過 JNI，你的原生函式可以：

- 產生 Java 物件（包括陣列和 **String**s），取得並更新 Java 物件值。
- 呼叫 Java 函式。

- 捕捉、擲出異常（exceptions）
- 載入 classes，並取得 class 的相關資訊。
- 執行動態時期型別檢驗（run-time type checking）

因此，可在一般 Java 程式中對 classes 和其物件進行的動作，也皆可於 原生函式中爲之。

原生函式（Native method）的呼叫

讓我們從一個簡單的例子開始。這是一個呼叫原生函式的 Java 程式。它會依序呼叫 C 標準程式庫的 **printf()** 函式。

第一個步驟便是撰寫一份 Java 程式碼，宣告原生函式和其引數：

```
//: appendixb:ShowMessage.java
public class ShowMessage {
  private native void ShowMessage(String msg);
  static {
    System.loadLibrary("MsgImpl");
    // Linux hack, if you can't get your library
    // path set in your environment:
    // System.load(
    //  "/home/bruce/tij2/appendixb/MsgImpl.so");
  }
  public static void main(String[] args) {
    ShowMessage app = new ShowMessage();
    app.ShowMessage("Generated with JNI");
  }
} ///:~
```

接續在原生函式宣告式之後的是個 **static** 區段，其內呼叫 **System.loadLibrary()**（你其實可以在任何地點呼叫它，但這種安排比較恰當）。**System.loadLibrary()** 會將 DLL 載入記憶體，並加以連結。DLL 必須位於你的系統程式庫路徑中。JVM 會自動補上副檔名，副檔名視平台的不同而異。

上述程式碼中你可以看到，呼叫 **System.load()** 的地方被註解掉了。這裡指定的是個絕對路徑，與環境變數無關。使用環境變數當然比較好，也更具移植性。但如果你無法得知環境變數，你可以將 **loadLibrary()** 呼叫動作註解掉，取消對 **System.load()** 的註解，並將路徑改爲你自己的目錄。

表頭檔產生器：javah

接下來請編譯你的 Java 原始檔。針對你所得到的 **.class** 檔執行 **javah**，並指定 **-jni** 選項（本書所附原始碼中的 makefile 會自動執行該動作）：

```
javah —jni ShowMessage
```

javah 會讀取 Java class 檔，針對其中每一個原生函式宣告，以 C 或 C++ 標頭檔的型式產生出函式原型宣告。以下便是本例的輸出結果 **ShowMessage.h**（略加編修以符合書頁編排）：

```
/* DO NOT EDIT THIS FILE
   - it is machine generated */
#include <jni.h>
/* Header for class ShowMessage */

#ifndef _Included_ShowMessage
#define _Included_ShowMessage
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     ShowMessage
 * Method:    ShowMessage
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL
Java_ShowMessage_ShowMessage
  (JNIEnv *, jobject, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

如你所見，透過前處理器指令 **#ifdef __cplusplus**，此檔案可被 C 或 C++ 編譯器加以編譯。第一個 **#include** 指令會插入 **jin.h**，那是個標頭檔，定義了檔案中其餘部份會用到的各種型別。**JNIEXPORT** 和 **JNICALL** 都是巨集，會被展開爲平台專有指令。**JNIEnv**、**jobject**、**jstring** 都是 JNI 資料型別的定義，我馬上會爲你介紹這些指令。

名稱重整（Name mangling）與函式標記（function signatures）

JNI 對於原生函式的名稱，規範了一種獨一無二的名稱重整（name mangling）手法。這很重要，因爲這是「虛擬機器將 Java 呼叫動作連結至原生函式」機制的一部份。基本上所有原生函式皆以 "Java" 起首，其後跟著「Java native 宣告」所棲身的 class 的名稱，再跟著 Java 原生函式名稱，其間各以底線字元（'_'）分隔。如果重載（overload）Java 原生函式，函式標記（signature）便會被附加於名稱之後；你可以在上例的原型宣告前的註解中，看到原生標記式（native signature）。如果你想獲得名稱重整和原生函式標記式的進一步資訊，請參考 JNI 說明文件。

實作出你自己的 DLL

接下來撰寫 C 或 C++ 原始碼，並在其中含入 **javah** 所產生的表頭檔，並實作出原生函式。編譯之後會產生一個動態連結程式庫（dynamic link library, DLL）。以上會因平台的不同而有所不同。例如下列程式碼會被編譯和連結，在 Windows 上產出一個 **MsgImpl.dll**，在 Unix/Linux 上產出一個 **MsgImpl.so**。你可以在本書所附光碟中找到和程式碼置放在一起的 makefile（亦可自 *www.BruceEckel.com* 免費下載），其中包含上述動作所需指令。

```
//: appendixb:MsgImpl.cpp
//# Tested with VC++ & BC++. Include path must
//# be adjusted to find the JNI headers. See
```

```
//# the makefile for this chapter (in the
//# downloadable source code) for an example.
#include <jni.h>
#include <stdio.h>
#include "ShowMessage.h"

extern "C" JNIEXPORT void JNICALL
Java_ShowMessage_ShowMessage(JNIEnv* env,
jobject, jstring jMsg) {
  const char* msg=env->GetStringUTFChars(jMsg,0);
  printf("Thinking in Java, JNI: %s\n", msg);
  env->ReleaseStringUTFChars(jMsg, msg);
} ///:~
```

傳入原生函式的那些引數，是原生函式用來和 Java 程式溝通的管道。第一引數的型別爲 **JNIEnv**，其中包含所有必要的 hook（扣鉤），讓你得以回呼（callback）JVM（下一節我會探討這個動作）。第二引數的意義依函式的類型而有不同。對 non-**static** 函式（如上例）來說，第二引數和 C++ 的 **this** 指標功用相同，而且類似 Java 的 **this**：是個 reference，指向原生函式的呼叫者（某個物件）。對 **static** 函式來說，它是個 reference，指向原生函式的實作者（以某個 **Class** 物件做爲代表）。

其餘引數代表的是傳入原生函式的各個 Java 物件。基本型別的傳遞也是透過這種方式，不過卻是採用傳值（*by value*）的方式。

接下來的數節，我會探討上述程式碼，焦點是如何在原生函式內存取、控制 JVM。

取用 JNI 函式：透過 JNIEnv 引數

所謂 JNI 函式，就是程式員可於原生函式中用來和 JVM 溝通的函式。一如你在上述例中所見，每個 JNI 原生函式都收到一個特殊引數（**JNIEnv** 引數）做爲其第一參數。那是一個指標，指向型別爲 **JNIEnv_** 的一個特殊的 JNI 資料結構。JNI 資料結構中有個元素也是指標，指向 JVM 所產生的

陣列，其內每個元素都是指標，分別指向各個 JNI 函式。提取（dereference）這些指標後（實際動作很簡單），便可自原生函式中呼叫 JNI 函式。每個 JVM 都提供有自己的 JNI 函式實作版本，永遠位於預先定義好的偏移位置（offset）上。

程式員透過 **JNIEnv** 引數便可取用許多函式。這些函式可以下述分類方式加以編組：

- 取得版本資訊
- 執行 class 和物件相關動作
- 處理指向 Java 物件的一些全域性 references 和區域性 references
- 存取 instance fields（譯註：隸屬於物件）和 static fields（譯註：隸屬於類別）
- 呼叫 instance 函式（譯註：non-static 函式）和 static 函式
- 執行字串動作和陣列動作
- 產生或處理 Java 異常

JNI 函式的個數很多，這裡並不一一探討。我會說明這些函式被運用時的背後基本原理。如果你想獲得更細節的資訊，請參考編譯器附的 JNI 說明文件。

如果你檢視 **jni.h** 表頭檔，你會看到 **#ifdef __cplusplus** 這個前處理器條件句，它說如果這個檔案被 C++ 編譯器編譯，**JNIEnv_** 結構體會被定義爲 class 形式，其中有許多 inline 函式，讓你得以透過簡單、熟悉的語法來取用 JNI 函式。例如前例中的這行 C++ 程式碼：

```
env->ReleaseStringUTFChars(jMsg, msg);
```

可以以這樣的 C 形式被呼叫：

```
(*env)->ReleaseStringUTFChars(env, jMsg, msg);
```

我想你會注意到，C 形式（當然）更爲複雜 — 你需要對 **env** 指標進行雙重提領（deferencing），而且你得將 **env** 做爲第一個引數傳給 JNI 函式。本附錄中的所有例子用的都是 C++ 形式。

存取 Java Strings

讓我們看看取用 JNI 函式的一個例子，請看 **MsgImpl.cpp** 中的程式碼。在這個程式中，型別爲 **JNIEnv** 的引數 **env** 被用來存取 **Java String**。**Java String** 的編碼採用 Unicode，所以如果你接收 **String**，而且想將它傳入沒有能力處理 Unicode 的函式中（例如 **printf()**），那麼你得先以 JNI 函式 **GetStringUTFChars()** 將它轉換爲 ASCII 字元。這個函式會接收 Java **String** 並將它轉換成 UTF-8 字元。（一個用以儲存 ASCII 值的 UTF-8 字元，寬度爲 8-bits，而如果用以儲存 Unicode，寬度爲 16-bits。如果原始字串全由 ASCII 組成，那麼所得結果也會是個 ASCII。）**GetStringUTFChars()** 是 **JNIEnv** 中的成員函式之一。若想取用 JNI 函式，我們可以使用 C++ 語法透過指標呼叫成員函式。這種形式可以取用所有 JNI 函式。

傳遞和運用 Java 物件

前一個例子中我將 **String** 傳入原生函式。你也可以將你自己撰寫的 Java 物件傳入原生函式，並於後者之內存取接收到的物件的各個資料成員（fields）和函式（methods）。

如果想傳遞物件，請在宣告原生函式時使用一般 Java 語法。下例中的 **MyJavaClass** 擁有一個 **public** 資料成員和一個 **public** 函式。**UserObjects** 則宣告一個原生函式，其中接收一個 **MyJavaClass** 物件。爲了觀察原生函式能否操控其引數，這個程式將引數的 **public** 資料成員設值，然後呼叫原生函式，然後印出 **public** 資料成員之值。

```java
//: appendixb:UseObjects.java
class MyJavaClass {
  public int aValue;
  public void divByTwo() { aValue /= 2; }
}

public class UseObjects {
  private native void
    changeObject(MyJavaClass obj);
  static {
    System.loadLibrary("UseObjImpl");
    // Linux hack, if you can't get your library
    // path set in your environment:
    // System.load(
    //"/home/bruce/tij2/appendixb/UseObjImpl.so");
  }
  public static void main(String[] args) {
    UseObjects app = new UseObjects();
    MyJavaClass anObj = new MyJavaClass();
    anObj.aValue = 2;
    app.changeObject(anObj);
    System.out.println("Java: " + anObj.aValue);
  }
} ///:~
```

編譯並執行 **javah** 後，你便可以開始實作原生函式。下例取得資料成員 ID 和函式 ID 後，透過 JNI 函式加以存取。

```cpp
//: appendixb:UseObjImpl.cpp
//# Tested with VC++ & BC++. Include path must
//# be adjusted to find the JNI headers. See
//# the makefile for this chapter (in the
//# downloadable source code) for an example.
#include <jni.h>
extern "C" JNIEXPORT void JNICALL
Java_UseObjects_changeObject(
JNIEnv* env, jobject, jobject obj) {
  jclass cls = env->GetObjectClass(obj);
  jfieldID fid = env->GetFieldID(
    cls, "aValue", "I");
  jmethodID mid = env->GetMethodID(
```

```
    cls, "divByTwo", "()V");
  int value = env->GetIntField(obj, fid);
  printf("Native: %d\n", value);
  env->SetIntField(obj, fid, 6);
  env->CallVoidMethod(obj, mid);
  value = env->GetIntField(obj, fid);
  printf("Native: %d\n", value);
} ///:~
```

C++ 函式內忽略 **this** 指標；它所接收的 **jobject**，是「Java 程式碼傳遞過來的 Java object reference」在原生端（native side）的形式。我們很單純地取得 **aValue**、列印其值、改變其值、呼叫 **jobject** 的 **divByTwo()**、再次列印其值。

如果想存取 Java 資料成員或函式，你得先使用 **GetFieldID()** 或 **GetMethodID()** 分別取得資料成員和函式的識別碼。這兩個函式接收(1) class object、(2) 一個字串，內含元素名稱、(3) 一個字串，代表型別相關資訊：要不就是資料成員的型別，要不就是函式的標記資訊（JNI 說明文件提供有更詳細的資訊）。你可運用這些函式回傳的識別碼來存取你所要的元素。這個方法似乎像在繞路，但畢竟你的原生函式並不知道 Java 物件的內部佈局，所以它必須透過 JVM 回傳的索引值來存取資料成員和函式s。這就允許不同的 JVMs 擁有不同的物件佈局，卻不致對你的原生函式造成影響。

如果你執行了 Java 程式，你會發現從 Java 端傳入的物件會被原生函式改變。但是傳遞的東西實際上是什麼呢？是個 C++ pointer 還是個 Java reference 呢？而且，在原生函式呼叫過程中，垃圾回收器會採取怎樣的處理方式呢？

當原生函式運行，垃圾回收器持續運作。但是這個機制保證在原生函式被呼叫過程中，你的物件不會被垃圾回收機制撿走。爲了確保此點，呼叫原生函式之前程式會先產出一個 local references，並於呼叫之後立刻摧毀。由於其壽命長於整個原生函式呼叫過程，所以你可以確定在原生函式執行期間，這些物件都可被合法使用。

由於每次原生函式被呼叫前都會產生上述 references，並於呼叫後被摧毀，所以你無法在你的原生函式中以 **static** 方式製作區域性副本。如果你所需的 reference，得在不同的函式叫用動作中持續存活，你得使用全域性 reference。然而全域性 reference 並不由 JVM 產生，不過你可以呼叫特定的 JNI 函式來產生代表某個區域性 reference 的全域性 reference。當你產生全域性 reference，你就得負責處理它所指向的物件的生命。全域性 reference（及其所指物件）會持續存活於記憶體中，直到程式員透過適當的 JNI 函式（類似 C **malloc()** 和 **free()**）加以釋放。

JNI 和 Java 異常

透過 JNI 可擲出或捕捉 Java 異常，並可列印其值或重擲，這些和在 Java 程式中都一樣。但程式員有權決定是否呼叫特定的 JNI 函式來處理異常。以下是一些用來異常處理的 JNI 函式：

- **Throw( )**
  擲出既有的異常物件。用於原生函式中重擲異常。
- **ThrowNew( )**
  產生新的異常物件並擲出。
- **ExceptionOccurred( )**
  判斷某個異常是否已被擲出且尚未被清除。
- **ExceptionDescribe( )**
  列印異常及堆疊追蹤結果（stack trace）。
- **ExceptionClear( )**
  清除懸而未決（pending）的異常。
- **FatalError( )**
  引發一個致命錯誤（fatal error）。不回返。

在上述這些函式中，你絕不能忽略 **ExceptionOccurred()** 和 **ExceptionClear()**。大多數 JNI 函式都可能產生異常，而沒有任何語言功能可供你取代 Java 的 **try** 區段。所以，你得在每次呼叫 JNI 函式之後一

律呼叫 **ExceptionOccurred()**，藉此檢驗某個異常是否被擲出。如果偵測到異常，你可能會選擇加以處理（而且可能會重新擲出）。無論如何你得確定異常最終會被清除。你可以在你的函式中使用 **ExceptionClear()** 完成清除動作，或在異常被重新擲出時於其他函式中加以清除。不論如何，一定得清除。

你必須確保異常被清除。如果不這麼做，當某個異常處於懸而未決（pending）的狀態時，呼叫 JNI 函式的結果將不可預期。異常期間還能夠安全運作的 JNI 函式非常少，即使有，也都是用來處理異常。

JNI 和多緒（threading）

由於 Java 是個多緒程式語言，所以可能有多個執行緒同時呼叫同一個原生函式。當某個原生函式執行未完，第二個執行緒加以呼叫，該原生函式的執行便有可能被暫停。一個原生函式是否爲 thread-safe（也就是不會在未受監督的情況下修改共用資料），完全取決於程式員。基本上你有兩種選擇：將原生函式宣告爲 **synchronized**，或採用其他策略來實作原生函式，確保正確的並行（concurrent）資料處理。

你不應該在執行緒之間傳遞彼此的 **JNIEnv** 指標，因爲此一指標所指的內部結構，在每一個執行緒中都會被配置一份，其中所含的資訊只對該執行緒有意義。

使用既存的程式碼

實作 JNI 原生函式（native methods）的最簡單方式，就是撰寫 Java class 中的原生函式原型，編譯該 class，並執行 **javah** 於該 **.class** 身上。但如果你手上已經有個十分龐大、已經存在的程式碼，而你希望從 Java 這一端呼叫這些程式碼，又該怎麼辦呢？修改 DLL 中的所有函式名稱，使它們符合 Java 的名稱重整（name mangling）規格，並不是可行的作法。最好的方法便是撰寫一個包裝用的 DLL，將它包裝於原程式外圍。Java 程式呼叫新的 DLL 中的函式，後者再呼叫原始 DLL 中的函式。這種解決方式不僅

被廣泛運用，大多數情況下你甚至非得這麼做不可，因爲在你能夠使用 object references 之前，得於這些 object references 之上呼叫 JNI 函式。

補充資訊

你可以在本書第一版（納於本書所附光碟，亦可於 *www.BruceEckel.com* 中免費下載）可以找到更多簡介性資料，包括 C（不是 C++）範例和 Microsoft 相關討論議題。你可以在 *java.sun.com* 中找到更多補充資訊（可在搜尋引擎中敲入關鍵字 "training & tutorials" 和（或）"native methods"）。《*Core Java 2, Volume II*》（by Horstmann and Cornell, Prentice-Hall, 2000）第 11 章對於原生函式有極佳的說明。

C: Java 編程準則（Programming Guidelines）

這份附錄所提供的建議，可以幫助你進行低階的程式設計，並幫助你寫碼。

當然，這些都只是一種方針而不是硬性規則。你應該視它們爲一種靈感來源。記住，某些情況下你需要加以變通或甚至打破規則。

設計

1. **優雅需要付出代價**。從短期利益來看，對某個問題提出優雅的解決方法，似乎可能花你更多的時間。但當它終於能夠正確執行並可輕易套用於新案例中，不需要花上數以時計，甚至以天計或以月計的辛苦代價時，你會看得到先前所花功夫的回報（即使沒有人可以衡量這一點）。這不僅給你一個可更容易開發和除錯的程式，也更易於理解和維護。這正是它在金錢上的價值所在。這一點有賴某種人生經驗才能夠了解，因爲當你努力讓某一段程式碼變得比較優雅時，你並不是處於一種具生產力的狀態下。但是，請抗拒那些催促你趕工的人們，因爲那麼做只會減緩你的速度罷了。

2. **先求能動，再求快**。即使你已確定某段程式碼極爲重要，而且是系統的重要瓶頸，這個準則依然成立。儘可能簡化設計，讓系統能夠先正確運作。如果程式的執行不夠快，再量測其效能。幾乎你總是會發現，你所認爲的「瓶頸」其實都不是問題所在。把你的時間花在刀口上吧。

3. 記住「各個擊破」的原理。如果你所探討的問題過於混雜，試著想像該問題的基本動作會是什麼，並假設這一小塊東西能夠神奇地處理掉最難的部份。這「一小塊」東西其實就是物件 — 請撰寫運用該物件的程式碼，然後檢視物件，並將其中困難的部份再包裝成其他物件，依此類推。

4. 區分 class 開發者和 class 使用者（使用端程式員）。class 使用者扮演著「客戶」角色，不需要（也不知道）class 的底層運作方式。class 開發者必須是 class 設計專家，並撰寫 class，使它能夠儘可能被大多數新手程式員所用，而且在程式中能夠穩當執行。一套程式庫只有在具備通透性的情況下，使用起來才會容易。

5. 當你撰寫 class 時，試著給予明確易懂的名稱，減少不必要的註解。你給客戶端程式員的介面，應該保持概念上的單純性。爲了這個目的，當函式的重載（overloading）適合製作出直覺、易用的介面時，請善加使用。

6. 你的分析和設計必須讓系統中的 classes 保持最少，且必須讓其 public interfaces 保持最少，以及讓這些 classes 和其他 classes 之間的關聯性（尤其是 base classes）保持最少。如果你的設計所得結果更甚於此，請問問自己，是否其中每一樣東西在整個程式生命期中都饒富價值？如果並非如此，那麼，維護它們會使你付出代價。開發團隊的成員都有不維護「無益於生產力提昇」的任何東西的傾向；這是許多設計方法論無法解釋的現象。

7. 讓所有東西儘量自動化。先撰寫測試用的程式碼（在你撰寫 class 之前），並讓它和 class 結合在一起。請使用 makefile 或類似工具，自動進行測試動作。透過這種方式，只要執行測試程式，所有的程式變動就可以自動獲得驗證，而且可以立即發現錯誤。由於你知道你的測試架構所具備的安全性，所以當你發現新的需求時，你會更勇於進行全面修改。請記住，程式語言最大的改進，是來自型別檢查、異常處理等機制所賦予的內建測試動作。但這些功能只能協助

你到達某種程度。開發一個穩固系統時，你得自己驗證自己的 classes 或程式的性質。

8. 在你撰寫 class 之前先寫測試碼，以便驗證你的 class 是否設計完備。如果你無法撰寫測試碼，你便無法知道你的 class 的可能長相。撰寫測試碼通常能夠顯現出額外的特性（features）或限制（constraints）— 它們並不一定總是能夠在分析和設計過程中出現。測試碼也可做爲展示 class 用法的範例程式。

9. 所有軟體設計上的問題，都可以透過「引入額外的概念性間接層（conceptual indirection）」加以簡化。這個軟體工程上的基礎法則[1] 是抽象化概念的根據，而抽象化概念正是物件導向程式設計的主要性質。

10. 間接層（indirection）應該要有意義（和準則 9 一致）。這裡所指的意義可以像「將共用程式碼置於唯一函式」這麼簡單。如果你加入的間接層（或抽象化、或封裝等等）不具意義，它可能就和沒有適當的間接層一樣糟糕。

11. 讓class 儘可能微小而無法切割（atomic）。賦予每個 class 單一而清楚的用途。如果你的 classes 或你的系統成長得過於複雜，請將複雜的 classes 切割成比較簡單的幾個 classes。最明顯的一個判斷指標就是 class 的大小：如果它很大，那麼它工作量過多的機會就可能很高，那就應該被切割。重新設計 class 的建議線索是：1)複雜的 switch 述句：請考慮運用多型（polymorphism）。2) 許多函式各自處理類型極爲不同的動作：請考慮切割爲多個不同的 classes。

---

[1]這是 Andrew Koenig 對我說的。

12. **小心冗長的引數列**（argument lists）。冗長的引數列會使函式的叫用動作不易撰寫、閱讀、維護。你應該試著將函式搬移到更適當的 class 中，並儘量以物件爲引數。

13. **不要一再重複**。如果某段程式碼不斷出現於許多 derived class 函式中，請將該段程式碼置於某個 base class 函式內，然後在 derived class 函式中呼叫。這麼做不僅可以省下程式碼空間，也可以讓修改該段程式碼的動作更易於進行。有時候，找出此種共通程式碼還可以爲介面增加實用功能。

14. **小心 switch 述句或成串的 if-else 子句**。通常這種情況代表所謂的 "type-check coding"，也就是說究竟會執行哪一段程式碼，乃是依據某種型別資訊來做抉擇（最初，確切型別可能不十分明顯）。你通常可以使用繼承和多型來取代此類程式碼；polymorphical method（多型函式）的叫用會自動執行此類型別檢驗，並提供更可靠更容易的擴充性。

15. **從設計觀點來看，請找出變動的事物，並使它們和不變的事物分離**。也就是說，找出系統中可能被你改變的元素，將它們封裝於 classes 中。你可以在《*Thinking in Patterns with Java*》（可免費下載於 *www.BruceEckel.com*）大量學習到這種觀念。

16. **不要利用 subclassing 來擴充基礎功能**。如果某個介面元素對 class 而言極重要，它應該被放在 base class 裡頭，而不是直到衍生（derivation）時才被加入。如果你在繼承過程中加入了函式，或許你應該重新思考整個設計。

17. **少就是多**。從 class 的最小介面開始發展，儘可能在解決問題的前提下讓它保持既小又單純。不要預先考量你的 class 被使用的所有可能方式。一旦 class 被實際運用，你自然會知道你得如何擴充介面。不過，一旦 class 被使用後，你就無法在不影響用戶程式碼的情況下縮減其介面。如果你要加入更多函式倒是沒有問題 — 不會影響既有的用戶程式碼，它們只需重新編譯即可。但即使新函式取代了舊函式的功能，也請你保留既有介面。如果你得透過「加入更多引數」的

方式來擴充既有函式的介面，請你以新引數寫出一個重載化的函式；透過這種方式就不會影響既有函式的任何用戶了。

18. 大聲念出你的 classes，確認它們符合邏輯。請讓 base class 和 derived class 之間的關係是 "*is-a*"（是一種），讓 class 和成員物件之間的關係是 "*has-a*"（有一個）。

19. 當你猶豫不決於繼承（inheritance）或合成（複合，composition）時，請你問問自己，是否需要向上轉型（upcast）爲基礎型別。如果不需要，請優先選擇合成（也就是使用成員物件）。這種作法可以消除「過多基礎型別」。如果你採用繼承，使用者會認爲他們應該可以向上轉型。

20. 運用資料成員來表示數值的變化，運用經過覆寫的函式（overrided method）來代表行爲的變化。也就是說，如果你找到了某個 class，帶有一些狀態變數，而其函式會依據這些變數值切換不同的行爲，那麼你或許就應該重新設計，在 subclasses 和覆寫後的函式（overrided methods）中展現行爲上的差異。

21. 小心重載（overloading）。函式不應該依據引數值條件式地選擇執行某一段程式碼。這種情況下你應該撰寫兩個或更多個重載函式（overloaded methods）。

22. 使用異常體系（exception hierarchies） — 最好是從 Java 標準異常體系中衍生特定的 classes，那麼，捕捉異常的人便可以捕捉特定異常，之後才捕捉基本異常。如果你加入新的衍生異常，原有的用戶端程式仍能透過其基礎型別來捕捉它。

23. 有時候簡單的聚合（aggregation）就夠了。飛機上的「旅客舒適系統」包括數個分離的元素：座椅、空調、視訊設備…等等，你會需要在飛機上產生許多這樣的東西。你會將它們宣告爲 private 成員並開發出一個全新的介面嗎？不會的，在這個例子中，元素也是 public 介面的一部份，所以你應該產生 public 成員物件。這些物件

具有它們自己的 private 實作，所以仍然是安全的。當然啦，簡單聚合並不是一個常被運用的解法，但有時候的確是。

24. 試著從客戶程式員和程式維護者的角度思考。你的 class 應該設計得儘可能容易使用。你應該預先考量可能有的變動，並針對這些可能的變動進行設計，使這些變動日後可輕易完成。

25. 小心「巨大物件併發症」。這往往是剛踏入 OOP 領域的程序式（procedural）程式員的一個苦惱，因為他們往往最終還是寫出一個程序式程式，並將它們擺放到一個或兩個巨大物件中。注意，除了 application framework（應用程式框架，譯註：一種很特殊的、大型 OO 程式庫，幫你架構程式本體）之外，物件代表的是程式中的觀念，而不是程式本身。

26. 如果你得以某種醜陋的方式來達成某個動作，請將醜陋的部份侷限在某個 class 裡頭。

27. 如果你得使用某種不可移植方式來達成某個動作，請將它抽象化並侷限於某個 class 裡頭。這樣一個「額外間接層」能夠防止不可移植的部份擴散到整個程式。這種作法的具體呈現便是 *Bridge* 設計樣式（design pattern）。

28. 物件不應僅僅只用來持有資料。物件也應該具有定義明確界限清楚的行為。有時候使用「資料物件」是適當的，但只有在通用型容器不適用時，才適合刻意以資料物件來包裝、傳輸一群資料項。

29. 欲從既有的 classes 身上產生新的 classes 時，請以複合（composition）為優先考量。你應該只在必要時才使用繼承。如果在複合適用之處你卻選擇了繼承，你的設計就滲雜了非必要的複雜性。

30. 運用繼承和函式覆寫機制來展現行為上的差異，運用 fields（資料成員）來展現狀態上的差異。這句話的極端例子，就是繼承出不同的 classes 來表現各種不同的顏色，而不使用 "color" field。

31. **當心變異性**（variance）。語意相異的兩個物件擁有相同的動作（或說責任）是可能的。OO 世界中存在著一種天生的引誘，讓人想要從某個 class 繼承出另一個 subclass，爲的是獲得繼承帶來的福利。這便是所謂「變異性」。但是，沒有任何正當理由足以讓我們強迫製造出某個其實並不存在的 superclass/subclass 關係。比較好的解決方式是寫出一個共用的 base class，它爲兩個 derived classes 製作出共用介面 — 這種方式會耗用更多空間，但你可以如你所盼望地從繼承機制獲得好處，而且或許能夠在設計上獲得重大發現。

32. **注意繼承上的限制**。最清晰易懂的設計是將功能加到繼承得來的 class 裡頭；繼承過程中拿掉舊功能（而非增加新功能）則是一種可疑的設計。不過，規則可以打破。如果你所處理的是舊有的 class 程式庫，那麼在某個 class 的 subclass 中限制功能，可能會比重新制定整個結構（俾使新 class 得以良好地相稱於舊 class）有效率得多。

33. **使用設計樣式**（design patterns）**來減少「赤裸裸無加掩飾的機能**（naked functionality）」。舉個例子，如果你的 class 只應該產出唯一一個物件，那麼請不要以不加思索毫無設計的手法來完成它，然後撰寫「只該產生一份物件」這樣的註解就拍拍屁股走人。請將它包裝成 *singleton*（譯註：一個有名的設計樣式，可譯爲「單件」）。如果主程式中有多而混亂的「用以產生物件」的程式碼，請找出類似 *factory method* 這樣的生成樣式（creational patterns），使你可用以封裝生成動作。減少「赤裸裸無加掩飾的機能」（naked functionality）不僅可以讓你的程式更易理解和維護，也可以阻止出於好意卻帶來意外的維護者。

34. **當心「因分析而導致的癱瘓**（analysis paralysis）」。請記住，你往往必須在獲得所有資訊之前讓專案繼續前進。而且理解未知部份的最好也最快的方式，通常就是實際前進一步而不只是紙上談兵。除非找到解決辦法，否則無法知道解決辦法。Java 擁有內建的防火牆，請讓它們發揮作用。你在單一 class 或一組 classes 中所犯的錯誤，並不會傷害整個系統的完整性。

35. 當你認爲你已經獲得一份優秀的分析、設計或實作時，請試著加以演練。將團隊以外的某些人帶進來 — 他不必非得是個顧問不可，他可以是公司其他團隊的成員。請那個人以新鮮的姿態審視你們的成果，這樣可以在尚可輕易修改的階段找出問題，其收穫會比因演練而付出的時間和金錢代價來得高。

實作（Implementation）

36. 一般來說，請遵守 Sun 的程式編寫習慣。你可以在以下網址找到相關文件：*java.sun.com/docs/codeconv/index.html*。本書儘可能遵守這些習慣。眾多 Java 程式員看到的程式碼，都是由這些習慣構成的。如果你固執地停留在過去的編寫風格中，你的（程式碼）讀者會比較辛苦。不論你決定採用什麼編寫習慣，請在整個程式中保持一致。你可以在 *home.wtal.de/software-solutions/jindent* 上找到一個用來重排 Java 程式的免費工具。

37. 無論使用何種編寫風格，如果你的團隊（或整個公司，那就更好了）能夠加以標準化，那麼的確會帶來顯著效果。這代表每個人都可以在其他人不遵守編寫風格時修改其作品，這是個公平的遊戲。標準化的價值在於，分析程式碼時所花的腦力較小，因而可以專心於程式碼的實質意義。

38. 遵守標準的大小寫規範。將 class 名稱的第一個字母應爲大寫。資料成員、函式、物件（references）的第一個字母應爲小寫。所有識別名稱的每個字都應該連在一塊兒，所有非首字的第一個字母都應該大寫。例如：

    ThisIsAClassName

    thisIsAMethodOrFieldName

    如果你在 **static final** 基本型別的定義處指定了常數初始式（constant initializers），那麼該識別名稱應該全爲大寫，代表一個編譯期常數。

    Packages 是個特例，其名稱皆爲小寫，即使非首字的字母亦是如

此。域名（com, org, net, edu 等等）皆應為小寫。（這是 Java 1.1 遷移至 Java 2 時的一項改變）

39. 不要自己發明「裝飾用的」private 資料成員名稱。通常這種名稱的形式是在最前端加上底線和其他字元。匈牙利命名法（Hungarian notation）是其中最差的示範。在這種命名法中，你得加入額外字元來表示資料的型別、用途、位置等等。彷彿你用的是組合語言（assembly langauge）而編譯器沒有提供任何協助似的。這樣的命名方式容易讓人混淆又難以閱讀，也不易推行和維護。就讓 classes 和 packages 來進行「名稱上的範圍制定（name scoping）」吧。

40. 當你撰寫具通用性的 class 時，請遵守正規形式（canonical form）。包括 **equals()**、**hashCode()**、**toString()**、**clone()**（實作出 **Cloneable**），並實作出 **Comparable** 和 **Serialiable** 等等。

41. 對於那些「取得或改變 private 欄位值」的函式，請使用 JavaBeans 的 "get"、"set"、"is" 等命名習慣，即使你當時不認為自己正在撰寫 JavaBean。這麼做不僅可以輕易以 Bean 的運用方式來運用你的 class，也是對此類函式的一種標準命名方式，使讀者更易於理解。

42. 對於你所撰寫的每一個class，請考慮為它加入 static public test()，其中含有 class 功能測試碼。你不需要移除該測試碼就可將程式納入專案。而且如果有所變動，你可以輕易重新執行測試。這段程式碼也可以做為 class 的使用範例。

43. 有時候你需要透過繼承，才得以存取 base class 的 protected 成員。這可能會引發對多重基礎類別（multiple base types）的認知需求。如果你不需要向上轉型，你可以先衍生新的 class 以便執行 protected 存取動作，然後在「需要用到上述 protected 成員」的所有 classes 中，將新 class 宣告為成員物件，而非直接繼承。

44. 避免純粹爲了效率考量而使用 final 函式。只有在程式能動但執行不夠快時，而且效能量測工具（profiler）顯示某個函式的叫用動作成爲瓶頸時，才使用 **final** 函式。

45. 如果兩個 classes 因某種功能性原因而產生了關聯（例如容器 containers 和迭代器 iterators），試著讓其中某個 class 成爲另一個 class 的內隱類別（inner class）。這不僅強調二者間的關聯，也是透過「將 class 名稱巢狀置於另一個 class 內」而使同一個 class 名稱在單一 package 中可被重複使用。Java 容器程式庫在每個容器類別中都定義了一個內隱的（inner）**Iterator** class，因而能夠提供容器一份共通介面。運用內隱類別的另一個原因是讓它成爲 **private** 實作物的一部份。在這裡，內隱類別會爲資訊隱藏帶來好處，而不是對上述的 class 關聯性提供助益，也不是爲了防止命名空間污染問題（namespace pollution）。

46. 任何時候你都要注意那些高度耦合（coupling）的 classes。請考慮內隱類別（inner classes）爲程式撰寫和維護帶來的好處。內隱類別的使用並不是要去除 classes 間的耦合，而是要讓耦合關係更明顯也更便利。

47. 不要成爲「過早最佳化」的犧牲品。那會讓人神經錯亂。尤其在系統建構初期，先別煩惱究竟要不要撰寫（或避免）原生函式（native methods）、要不要將某些函式宣告爲 final、要不要調校程式碼效率等等。你的主要問題應該是先證明設計的正確性，除非設計本身需要某種程度的效率。

48. 讓範圍（作用域，scope）儘可能愈小愈好，這麼一來物件的可視範圍和壽命都將儘可能地小。這種作法可降低「物件被用於錯誤場所，因而隱藏難以察覺的臭蟲」的機會。假設你有個容器，以及一段走訪該容器的程式片段。如果你複製該段程式碼，將它用於新的容器身上，你可能會不小心以舊容器的大小做爲新容器的走訪上限值。如果舊容器已不在存取範圍內，那麼編譯期便可找出這樣的錯誤。

49. 使用 Java 標準程式庫提供的容器。請熟悉它們的用法，你將因此大幅提昇你的生產力。請優先選擇 **ArrayList** 來處理序列（sequences），選擇 **HashSet** 來處理集合（sets）、選擇 **HashMap** 來處理關聯式陣列（associative arrays），選擇 **LinkedList**（而不是 **Stack**）來處理 stacks 和 queues。

50. 對一個強固的（robust）程式而言，每一個組成都必須強固。請在你所撰寫的每個 class 中運用 Java 提供的所有強固提昇工具：存取權限、異常、型別檢驗…等等。透過這種方式，你可以在建構系統時安全地移往抽象化的下一個層次。

51. 寧可在編譯期發生錯誤，也不要在執行期發生錯誤。試著在最靠近問題發生點的地方處理問題。請優先在「擲出異常之處」處理問題，並在擁有足夠資訊以處理異常的最接近處理常式（handler）中捕捉異常。請進行現階段你能夠對該異常所做的處理；如果你無法解決問題，應該再次擲出異常。

52. 當心冗長的函式定義。函式應該是一種簡短的、「描述並實作 class 介面中某個可分離部份」的功能單元。過長且複雜的函式不僅難以維護，維護代價也高。或許它嘗試做太多事情了。如果你發現這一類函式，代表它應該被切割成多個函式。這種函式也提醒你或許得撰寫新的 class。小型函式同樣能夠在你的 class 中被重複運用。（有時候函式必須很大才行，但它們應該只做一件事情）

53. 儘可能保持 "private"。一旦你對外公開了程式庫的概況（method、class、或 field），你便再也無法移除它們。因爲如果移除它們，便會破壞某個現有的程式碼，使得它們必須重新被編寫或重新設計。如果你只公開必要部份，那麼你便可以改變其他東西而不造成傷害。設計總是會演化，所以這是個十分重要的自由度。透過這種方式，實作碼的更動對 derived class 造成的衝擊會降到最低。在多緒環境下，私密性格外重要 — 只有 private 欄位可受保護而不被 un-synchronized（未受同步控制）的運用所破壞。

54. 大量運用註解，並使用 javadoc 的「註解文件語法」來產生程式的說明文件。不過，註解應該賦予程式碼真正的意義；如果只是重申程式碼已經明確表示的內容，那是很煩人的。請注意，通常 Java class 和其函式的名稱都很長，爲的便是降低註解量。

55. 避免使用「魔術數字」，也就是那種寫死在程式碼裡頭的數字 — 如果你想改變它們，它們就會成爲你的惡夢，因爲你永遠都沒有辦法知道 "100" 究竟是代表「陣列大小」或其他東西。你應該產生具描述性的常數名稱，並在程式中使用該常數名稱。這使程式更易於理解也更易於維護。

56. 撰寫建構式時，請考慮異常狀態。最好情境下，建構式不執行任何會擲出異常的動作。次佳情境下，class 只繼承自（或合成自）強固的（robust）classes，所以如有任何異常被擲出，並不需要清理。其他情況下，你就得在 **finally** 子句中清理合成後的 classes。如果某個建構式一定會失敗，適當的動作就是擲出異常，使呼叫者不至於盲目認爲物件已被正確產生而繼續執行。

57. 如果你的 class 需要在「用戶程式員用完物件」後進行清理動作，請將清理動作放到單一而定義良好的函式中，最好令其名稱爲 **cleanup()** 以便能夠將用途告訴他人。此外請將 **boolean** 旗標放到 class 中，用以代表物件是否已被清理，使 **finalize()** 得以檢驗其死亡條件（請參考第 4 章）。

58. ***finalize()*** 只可用於物件死亡條件的檢驗（請參考第 4 章），俾有益於除錯。特殊情況下可能需要釋放一些不會被垃圾回收器回收的記憶體。因爲垃圾回收器可能不會被喚起處理你的物件，所以你無法使用 **finalize()** 執行必要的清理動作。基於這個原因，你得撰寫自己的「清理用」函式。在 class **finalize()**中，請檢查確認物件的確已被清理，並在物件尚未被清理時，擲出衍生自 **RuntimeException** 的異常。使用這種架構前，請先確認 **finalize()** 在你的系統上可正常運作（這可能需要呼叫 **System.gc()** 來確認）。

59. 如果某個物件在某個特定範圍（scope）內必須被清理（cleaned up），而不是被垃圾回收機制回收，請使用以下方法：將物件初始化，成功後立刻進入擁有 **finally** 子句的一個 **try** 區段內。**finally** 子句會引發清理動作。

60. 當你在繼承過程中覆寫了 **finalize()**，請記得呼叫 **super.finalize()**。但如果你的「直接上一層 superclass」是 **Object**，就不需要這個動作。你應該讓 **super.finalize()** 成爲被覆寫（overridden）之 **finalize()** 的最後一個動作而不是第一個動作，用以確保 base class 的組件在你需要它們的時候仍然可用。

61. 當你撰寫固定大小的物件容器，請將它們轉換爲陣列 — 尤其是從某個函式回傳此一容器時。透過這種方式，你可以獲得陣列的「編譯期型別檢驗」的好處，而且陣列接收者可能不需要「先將陣列中的物件加以轉型」便能加以使用。請注意，容器程式庫的 base class（*Java.util.Collection*）具有兩個 **toArray()**，能夠達到這個目的。

62. 在 interface（介面）和 abstract class（抽象類別）之間，優先選擇前者。如果你知道某些東西即將被設計爲一個 base class，你的第一選擇應該是讓它成爲 **interface**；只有在一定得放進函式或資料成員時，才應該將它改爲 **abstract** class。**interface** 只和「用戶端想進行什麼動作」有關，class 則比較把重心放在實作細節上。

63. **在建構式中只做唯一必要動作：將物件設定至適當狀態。**避免呼叫其他函式（除了 **final** 函式），因爲這些函式可能會被其他人覆寫因而使你在建構過程中得到不可預期的結果（請參考第 7 章以取得更詳細的資訊）。小型而簡單的建構式比較不可能擲出異常或引發問題。

64. **爲了避免一個十分令人洩氣的經驗，請確認你的 classpath 中的每個名稱，都只有一個未被放到 packages 裡頭的 class。**否則編譯器會先找到另一個名稱相同的 class，並回報錯誤訊息。如果你懷疑你的 classpath 出了問題，試著從 classpath 中的每個起點搜尋同名的 **.class** 檔案。最好還是將所有 classes 都放到 packages 裡頭。

65. **留意一不小心犯下的重載（overloading）錯誤。**如果你覆寫 base class 函式時沒有正確拼寫其名稱，那麼便會增加一個新的函式，而不是覆寫原有的函式。但是這種情況完全合法，所以你不會從編譯器或執行期系統得到任何錯誤訊息 — 你的程式碼只是無法正確作用，如此而已。

66. **當心過早最佳化。**先讓程式動起來，再讓它快 — 但只有在你必須（也就是說只有在程式被證明在某段程式碼上遭遇效能瓶頸）時才這麼做。除非你已經使用效能量測工具（profiler）找出瓶頸所在，否則你可能只是在浪費你的時間。效能調校的「隱藏成本」便是讓你的程式碼變得更不可讀、更難維護。

67. **記住，程式碼被閱讀的時間多於被撰寫的時間。**清晰的設計能夠製作出易懂的程式。註解、細節說明、範例都是無價的。這些東西能夠幫助你和你的後繼者。如果沒有其他資訊，那麼從 Java 線上文件找出一些有用的資訊時，你所遭遇的挫敗應該足以讓你相信這一點。

D：資源

軟體

從 *java.sun.com* 取得 **JDK**（Java Development Kid）。即使你選擇其他廠商的開發環境，當你懷疑可能是編譯器出錯的時候，手邊有個 JDK 還是不錯的。JDK 是檢驗標準，而且如果 JDK 中有個臭蟲，這隻臭蟲廣爲人知的機會應該很高。

從 *java.sun.com* 取得 Java 說明文件 HTML 版本。我看過的每一本介紹標準 Java 程式庫的書，不是跟不上最新資訊，就是有所遺漏。雖然 Sun 的這份文件存在一些小錯誤，而且有些項目的資訊過少，但起碼所有的 classes 和函式都在裡頭。相對於紙本，人們初期對線上資源的運用或許並不那麼稱手，但的確值得花上一些時間加以克服。無論如何請先打開它（HTML 文件），那麼至少可以先取得整個概觀。如果這時候你還是無法理解，再回到紙本書吧。

書籍

Thinking in Java, 1st Edition. 本書所附光碟裡頭就有此書經過完全索引並以顏色標註語法的一個 HTML 版本。你也可以從 *www.BruceEckel.com* 免費下載。此書涵蓋比較舊的、不適合被放進第二版的一些材料。

Core Java 2, by Horstmann & Cornell, Volume I：Fundamentals (Prentice-Hall, 1999). Volume II：Advanced Features, 2000。資訊量龐大又完整。每當我需要搜尋某些答案時，總會先翻開這兩本書。當你讀完 *Thinking in Java* 後還需要撒出更大的網時，我推薦這兩本書。

Java in a Nutshell: A Desktop Quick Reference, 2nd Edition, by David Flanagan (O'Reilly, 1997)。Java 線上文件的一份簡潔摘要。從個人

觀點來說，我偏好線上瀏覽 *java.sun.com* 的各種文件，尤其因爲它們的內容常會更動。不過許多人還是偏好紙本文件，而且這樣也比較有付費的質感。本書提供比線上文件更豐富的討論內容。

The Java Class Libraries: An Annotated Reference, by Patrick Chan and Rosanna Lee (Addison-Wesley, 1997)。線上文件應該這樣：有足夠的說明讓讀者可以實際運用。*Thinking in Java* 的一位技術審稿者這麼說：『如果我只能有一本 Java 書籍，那就是上面這本了（好啦，當然還有你那本☺）』。我並不像這位審稿者對這本書那麼感到興奮。這是一本很龐大很昂貴的書，其中提供的範例品質並不能滿足我。但是當你遇到某個難解問題時，到這本書來找尋答案是個不錯的主意。而且本書似乎也較 *Java in a Nutshell* 有深度（以及厚度）。

Java Network Programming, by Elliotte Rusty Harold (O'Reilly, 1997)。直到閱讀了這本書，我才開始了解 Java 的網絡機制。我發現作者的網站 Café au Lait 在 Java 開發上提供了讓人倍感興奮、有主見、最即時的觀點，而且不帶任何廠商色彩。作者定期更新網站內容，使網站內容得以跟上 Java 快速變動的各種消息。請看 *matalab.unc.edu/javafaq/*。

JDBC Database Access with Java, by Hamilton, Cattell & Fisher (Addison-Wesley, 1997)。如果你對 SQL 和資料庫一無所知，這是一本極好的入門書。它同時也含括了某些 API 細節介紹，以及「帶有註釋的」API 參考資料（這也是線上參考文件所應具備的）。本書缺點和 ***The Java Series***（唯一由 JavaSoft 認可的系列書籍）一樣，過於粉飾太平，只講 Java 美好的一面。該系列中你找不到任何負面資訊。

Java Programming with CORBA, by Andreas Vogel & Keith Duddy (John Wiley & Sons, 1997)。附有三種 Java ORBs（Visibroker、Orbix、Joe）程式範例，對 CORBA 課題有深入的探討。

Design Patterns, by Gamma, Helm, Johnson & Vlissides (Addison-Wesley, 1995)。在程式設計領域中帶動設計樣式（design patterns）風潮的一本種子書。

Practical Algorithms for Programmers, by Binstock & Rex (Addison-Wesley, 1995)。書中演算法以 C 呈現，很容易轉換爲 Java。每個演算法都有極完整的解說。

分析 & 設計

Extreme Programming Explained, by Kent Beck (Addison-Wesley, 2000)。我愛這本書。是的，我傾向於採用根本方法來解決問題，但我總認爲應該會有更不一樣、更好的程式開發過程，我認爲 XP（eXtreme Porgramming）已經很逼近這個境界了。對我而言，唯一有相似影響力的另一本書就是 ***PeopleWare***（待會兒爲你介紹），主要探討環境和團體文化的處理。***eXtreme Programming Explained*** 探討的是程式設計，而且顛覆一般人所知的絕大多數方法 — 甚至顛覆最新的「研究結果」。他們甚至誇張到認爲任何描述只要別花你太多時間，而且你願意將它們丟掉，就可以採用 — 喔，你會發現該書封面並沒有 "UML stamp of approval"（UML 認同標籤）。我會以某家公司是否採用 XP 來決定是否爲他們工作。這是一本小書，章節少，讀起來不費力，而且能夠激勵思考。你會開始想像自己彷彿工作在這樣的氛圍當中，它會帶給你全新視野。

UML Distilled, 2nd Edition, by Martin Fowler (Addison-Wesley, 2000). 初次接觸 UML 時你大概會有怯步的感覺。因爲裡頭的圖示法和細節實在太多了。根據 Fowler 的說法，大部份內容都非必要，所以他直接討論本質。對大多數專案來說，你只需要知道少數幾種圖示工具就夠了。Fowler 的目的便是提供一份好的設計，而不是去考慮所有達成這一份好設計所需要的所有工具。這是一本優秀、輕薄、易讀的書籍；如果你需要了解 UML，它是你應該讀的第一本。

UML Toolkit, by Hans-Erik Eriksson & Magnus Penker, (John Wiley & Sons, 1997)。本書解說 UML 本身及其運用方式。並以 Java 進行個例探討。書附光碟內含 Java 程式碼和 Rational Rose 陽春版。在 UML 以及「如何運用 UML 建構真實系統」上，這是一份出色的簡介性書籍。

The Unified Software Development Process, by Ivar Jacobsen, Grady Booch, and James Rumbaugh (Addison-Wesley, 1999)。我原本做好了不喜歡這本書的心理準備。此書似乎具備煩人的大學教科書所具備的

一切必要條件。我甚至帶點竊喜地發現，此書有少數幾個觀念好像連作者也不甚明白似的。本書不僅清晰，甚至帶點閱讀樂趣。本書最好的一點是，整個過程帶來許多實用價值。這不僅是 eXtreme Programming 也是 UML 的駭人力量之一。縱使你無法接受 XP，由於大多數人都已搭上「UML 就是好」的時尚流行（不論他們實際經驗爲何），所以你或許可以接受本書。我認爲此書應該是 UML 旗艦書品。當你讀完 Fowler 的 *UML Distilled* 後，如果想知道更詳細的資訊，可以選擇這本書。

選擇任何方法（論）之前，先從那些非狂熱份子身上獲得一些看法，會有幫助。人們往往在尚未真正了解某種方法，或尚未知道某種方法能爲你做些什麼之前，就輕率選擇了它們。「其他人正在使用」似乎是個令人信服的理由，但人們都有一種奇怪的心理怪癖：如果人們想要相信某個東西真能解決問題，他們就會去嘗試（這種實驗態度很好），但如果它不能解決問題，他們便可能加倍努力並開始大聲宣稱，他們發現了很偉大的東西（這種拒絕承認的態度不好）。這裡的假設是，或許有了一些人和你在同一艘船上，你就不感到孤單了，即使那艘船駛往不知名的地方（甚至正在下沉）。

我並不是暗示所有方法論都沒有價值，而是提醒你應該使用一些精神上的工具來武裝自己，這個工具能夠幫助你繼續停留在實驗模式（『這種方法不可行，讓我們試試其他方法』）而脫離否認模式（『噢不，這其實不是問題。一切都是那麼美好，我們不需要改變』）。我認爲在你選擇某種方法（論）之前，先讀過下列幾本書，會給你帶來這樣子的工具。

Software Creativity, by Robert Glass (Prentice-Hall, 1995)。在討論整個方法論問題的觀點上，這是我見過最好的一本書。本書集合了 Glass 所撰寫或取得（P.J. Pluger 是其中一位作者）的許多短文和論文，這些文章反映出他多年來對這個課題的思考和研究。這些文章十分有趣，而且長度恰好；既非信手塗鴉，也不會讓你感到無聊。作者不是在放煙霧彈，書中參考的其他論文和研究報告數以百計。所有程式員和管理者在陷入方法論的泥沼前，都應該好好閱讀這本書。

Software Runaways: Monumental Software Disasters, by Robert Glass (Prentice-Hall, 1997)。這本書最棒的地方就是，它將我們帶到我們沒有討論的最前線去：有多少專案不僅失敗，而且一敗塗地。我發現大多數人仍然認爲「這不可能發生在我身上」（或是「這不會再重演」），而我認爲這會使你處於劣勢。把「任何事都可能出錯」牢記心中，你便比較能夠站在較好的位置來修正發生的問題。

Peopleware, 2nd Edition, by Tom Demarco and Timothy Lister (Dorset House, 1999)。雖然這些人的背景是軟體開發，但這本書所討論的是一般專案和團隊，重心擺在人與人的需求，而不是技術與技術的需求。作者所討論的是如何建立一個讓人們能夠快樂工作並且有高生產力的環境，而不是討論這些人應該遵守那些規則才能夠成爲稱職的機器零件。我認爲後一種態度正是程式員「面對某種方法時陽奉陰違」的最大肇因。

Complexity, by M. Mitchell Waldrop (Simon & Schuster, 1992)。這本書記錄了一群來自不同領域的科學家，聚集於新墨西哥州聖塔菲亞（Santa Fe），一起討論他們各自學科領域無法解決的現實問題（經濟學裡的股市問題、生物學裡的生命初始問題、社會學裡的人類行爲問題…等等）。藉著跨越物理、經濟、化學、數學、計算機科學、社會學、以及其他種種學科的方式，針對這些問題發展出跨學科解決方案。更重要的是，思考這類極複雜問題的另一種方式正在成形：捨棄「數學決定論」和「能以方程式預測所有行爲」的錯誤認知，邁向「先觀察、找出模式、試著以任何可能的手段模擬」的方式。舉個例子，此書記錄了遺傳演算法（genetic algorithms）的面世。我相信這種思考方式對我們觀察愈來愈複雜的軟體專案十分有用。

Python

Learning Python, by Mark Lutz and David Ascher (O'Reilly, 1999)。一份很好的針對程式員的介紹。它所介紹的東西短時間內就成了我最喜歡的程式語言。和 Java 搭配使用更是一級棒。本書還包括對 JPython 的介紹。透過 JPython，你可以將 Java 和 Python 合併於同一個程式裡（JPython 直譯器會被編譯成純粹的 Java bytecodes；所以不需要加入任何特別的東

西就可以達成這樣的目的）。這個語言的相關組織承諾爲我們帶來最大的可能性。

我的著作

以下以出版順序排列。並非所有書籍都能在市面上找到。

Computer Interfacing with Pascal & C, （透過 Eisys imprint 自行印製, 1998）只能於 *www.BruceEckle.com* 取得。這是在 CP/M 稱王 DOS 崛起的時代，一本帶有電子學基礎的簡介書。我過去常常使用高階語言透過電腦平行埠進行控制，完成各種電子專案。本書內容改寫自我在 *Micro Cornucopia* 雜誌上的專欄文章。那本雜誌是我的第一個（也是最好的一個）專欄舞台。但是，唉，在 Internet 出現的很久之前 *Micro Cornucopia* 雜誌就已經消失了。本書帶給我極滿意的出版經驗。

Using C++, (Osborne/McGraw-Hill, 1989)。我的第一本 C++ 書籍。本書已經絕版，被其後繼者 *C++ Inside & Out* 取代。

C++ Inside & Out, (Osborne/McGraw-Hill, 1993)。就如上段所提，本書實際上是 *Using C++*的第二版。本書內容已經相當精確，但 1992 左右我以 *Thinking in C++*取代之。你可以在 *www.BruceEckel.com* 中找到更多本書資訊，也可以下載取得原始碼。

Thinking in C++, 1st Edition, (Prentice-Hall, 1995).

Thinking in C++, 2nd Edition, Volume 1, (Prentice-Hall, 2000). 可自 *www.BruceEckel.com* 下載。

Black Belt C++, the Master's Collection, 由 Bruce Eckel 編纂而成，M&T Books 出版, 1994。本書已絕版，其內容乃是收集我擔任「軟體開發會議（Software Development Conference）」大會主席期間，許多 C++ 傑出人物的投稿文章。本書封面促使我想要控制日後所有我的書籍的封面設計。

Thinking in Java, 1st Edition, (Prentice-Hall, 1998)。這是你手上這本書的第一版，贏得 *Software Development Magazine* 生產力獎、*Java Developer's Journal* 編輯優選獎、*JavaWorld Reader* 最優選書籍獎。可於 *www.BruceEckel.com* 下載取得。

索引

請注意，某些名稱會以大寫型式重複一次。以下遵循 Java 風格，也就是大寫名稱代表 Java classes，小寫名稱代表一般概念。

- · 139
! · 143
!= · 141; operator · 1025
& · 146
&& · 143
&= · 147
@deprecated · 128
[ ]: indexing operator [ ] · 231
^ · 146
^= · 147
| · 146
|| · 143
|= · 147
'+': operator + for String · 1054
+ · 139
< · 141
<< · 147
<<= · 147
<= · 141
== · 141; operator · 1025; vs. equals( ) · 645
> · 141
>= · 141
>> · 147
>>= · 147

A

abstract: class · 326; inheriting from an abstract class · 326; vs. interface · 356
abstract keyword · 327
Abstract Window Toolkit (AWT) · 689
AbstractButton · 734
abstraction · 30
AbstractSequentialList · 502
AbstractSet · 461
accept( ) · 909
access: class · 263; control · 243, 267; inner classes & access rights · 376; package access and friendly · 255; specifiers · 36, 243, 255; within a directory, via the default package · 257
action command · 765
ActionEvent · 766, 814
ActionListener · 712
actor, in use cases · 77
adapters: listener adapters · 729
add( ), ArrayList · 450
addActionListener( ) · 811, 857
addChangeListener · 771
addition · 137
addListener · 722
addXXXListener( ) · 723
Adler32 · 608
aggregate array initialization · 231
aggregation · 37
aliasing · 136; and String · 1054; during a method call · 1014
align · 697
alphabetic vs. lexicographic sorting · 436
AlreadyBoundException · 978
analysis: and design, object-oriented · 71; paralysis · 72; requirements analysis · 75
AND: bitwise · 154; logical (&&) · 143
anonymous inner class · 370, 576, 709, 875; and constructors · 375
anonymous inner class, and table-driven code · 502
applet · 692; advantages for client/server systems · 693; align · 697; and packages · 699; archive tag, for HTML and JAR files · 793; classpath · 699; codebase · 697; combined applets and applications · 700; displaying a Web page from within an applet · 923; name · 697; packaging applets in a JAR file to optimize loading · 793; parameter · 697;

placing inside a Web page · 695; restrictions · 692
Applet: combined with application · 839; initialization parameters · 839
appletviewer · 698
application: application builder · 800; application framework · 394; combined applets and applications · 700; combined with Applet · 839; windowed applications · 700
application framework, and applets · 694
archive tag, for HTML and JAR files · 793
argument: constructor · 193; final · 298, 577; passing a reference into a method · 1014; variable argument lists (unknown quantity and type of arguments) · 235
array · 407; associative array · 477; associative array, Map · 442; bounds checking · 232; comparing arrays · 431; copying an array · 429; dynamic aggregate initialization syntax · 412; element comparisons · 431; first-class objects · 409; initialization · 231; length · 232, 409; multidimensional · 236; of objects · 409; of primitives · 409; return an array · 413
ArrayList · 456, 463, 467, 500, 505; add( ) · 450; and deep copying · 1030; get( ) · 450, 456; size( ) · 451; type-conscious ArrayList · 454; used with HashMap · 652
Arrays class, container utility · 415
Arrays.asList( ) · 519
Arrays.binarySearch( ) · 437
Arrays.fill( ) · 428
assigning objects · 134
assignment · 134
associative array · 439, 477
associative arrays (Maps) · 442
auto-decrement operator · 139
auto-increment operator · 139
automatic type conversion · 273
available( ) · 598

B

bag · 440
base: types · 39
base 16 · 156
base 8 · 156
base class · 260, 275, 315; abstract base class · 326; base-class interface · 320; constructor · 332; constructors and exceptions · 281; initialization · 278
Basic: Microsoft Visual Basic · 800
basic concepts of object-oriented programming (OOP) · 29
BASIC language · 92
BasicArrowButton · 735
beanbox Bean testing tool · 817
BeanInfo: custom BeanInfo · 818
Beans: and Borland's Delphi · 800; and Microsoft's Visual Basic · 800; and multithreading · 854; application builder · 800; beanbox Bean testing tool · 817; bound properties · 818; component · 801; constrained properties · 818; custom BeanInfo · 818; custom property editor · 818; custom property sheet · 818; events · 801; EventSetDescriptors · 808; FeatureDescriptor · 818; getBeanInfo( ) · 805; getEventSetDescriptors( ) · 808; getMethodDescriptors( ) · 808; getName( ) · 808; getPropertyDescriptors( ) · 808; getPropertyType( ) · 808; getReadMethod( ) · 808; getWriteMethod( ) · 808; indexed property · 818; Introspector · 805; JAR files for packaging · 816; manifest file · 816; Method · 808; MethodDescriptors · 808; naming convention · 802; properties · 801; PropertyChangeEvent · 818; PropertyDescriptors · 808; ProptertyVetoException · 818; reflection · 801, 804; Serializable · 814; visual programming · 800
Beck, Kent · 1093
Bill Joy · 141
binary: numbers · 156; operators · 146
binary numbers, printing · 150
binarySearch( ) · 437
bind( ) · 976
binding: dynamic binding · 316; dynamic, late, or run-time binding · 311; early · 45; late · 45; late binding · 316; method call binding · 315; run-time binding · 316
BitSet · 522
bitwise: AND · 154; AND operator (&) · 146; EXCLUSIVE OR XOR (^) · 146; NOT ~ ·

146; operators · 146; OR · 154; OR operator (|) · 146
bitwise copy · 1024
blank final · 297
blocking: and available( ) · 598; and threads · 859; on I/O · 869
Booch, Grady · 1093
book: errors, reporting · 23; updates of the book · 22
boolean: operators that won't work with boolean · 141
Boolean · 169; algebra · 146; and casting · 155; vs. C and C++ · 144
BorderLayout · 713
Borland · 820; Delphi · 800
bound properties · 818
bounds checking, array · 232
Box, for BoxLayout · 718
BoxLayout · 717
break keyword · 175
browser: class browser · 263
BufferedInputStream · 586
BufferedOutputStream · 588
BufferedReader · 563, 591, 597
BufferedWriter · 591, 599
business objects/logic · 796
button: creating your own · 730; radio button · 750
button, Swing · 706
ButtonGroup · 736, 750
buttons · 734
ByteArrayInputStream · 582
ByteArrayOutputStream · 583

C

C/C++, interfacing with · 1065
C++ · 141; copy constructor · 1042; Standard Container Library aka STL · 440; strategies for transition to · 93; templates · 455; vector class, vs. array and ArrayList · 408; why it succeeds · 91
callback · 432, 575, 708
callbacks: and inner classes · 391
capacity, of a HashMap or HashSet · 491
capitalization: Java capitalization style source-code checking tool · 645; of package names · 116
case statement · 183
cast · 47, 201, 661; and containers · 450; and primitive types · 170; from float or double to integral, truncation · 186; operators · 154
catch: catching an exception · 534; catching any exception · 543; keyword · 535
CD ROM for book · 20
CGI: Common-Gateway Interface · 948
change: vector of change · 397
CharArrayReader · 590
CharArrayWriter · 590
check box · 748
CheckedInputStream · 606
CheckedOutputStream · 606
Checksum · 608
class · 32, 262; abstract class · 326; access · 263; anonymous inner · 709; anonymous inner class · 370, 576, 875; anonymous inner class and constructors · 375; base class · 260, 275, 315; browser · 263; class hierarchies and exception handling · 567; class literal · 664, 669; creators · 35; defining the interface · 88; derived class · 315; equivalence, and instanceof/isInstance( ) · 672; final classes · 301; inheritance diagrams · 293; inheriting from an abstract class · 326; inheriting from inner classes · 384; initialization & class loading · 304; initialization of data members · 220; initializing members at point of definition · 221; initializing the base class · 278; inner class · 365; inner class nesting within any arbitrary scope · 372; inner classes · 799; inner classes & access rights · 376; inner classes and overriding · 385; inner classes and super · 385; inner classes and Swing · 722; inner classes and upcasting · 368; inner classes in methods & scopes · 370; inner classes, identifiers and .class files · 387; instance of · 31; intializing the derived class · 278; keyword · 38; loading · 305; member initialization · 273; multiply-nested · 383; order of initialization · 223; private inner classes · 397; public class, and compilation units · 245; read-only classes · 1047; referring to the outer class object in an inner class · 381; static inner classes · 379; style of creating classes · 262; subobject · 278

Class · 737; Class object · 633, 662, 848; forName( ) · 664, 727; getClass( ) · 544; getConstructors( ) · 681; getInterfaces( ) · 676; getMethods( ) · 681; getName( ) · 677; getSuperclass( ) · 676; isInstance · 671; isInterface( ) · 677; newInstance( ) · 676; printInfo( ) · 677; RTTI using the Class object · 674
Class object · 227
ClassCastException · 345, 666
classpath · 248, 699; and rmic · 979
class-responsibility-collaboration (CRC) cards · 79
cleanup: and garbage collector · 283; performing · 209; with finally · 554
cleanup, guaranteeing with finalize( ) · 214
client programmer · 35; vs. library creator · 243
client, network · 907
clipboard: system clipboard · 790
clone( ) · 1021; and composition · 1027; and inheritance · 1034; Object.clone( ) · 1025; removing/turning off cloneability · 1036; super.clone( ) · 1025, 1041; supporting cloning in derived classes · 1036
Cloneable interface · 1022
CloneNotSupportedException · 1024
close( ) · 597
closure, and inner classes · 391
code: calling non-Java code · 1065; coding standards · 22, 1077; organization · 255; re-use · 271
codebase · 697
Collection · 440
collection class · 407
Collections · 511
Collections.enumeration( ) · 520
Collections.fill( ) · 443
Collections.reverseOrder() · 434
collision: name · 250
collisions, during hashing · 488
com.bruceeckel.swing · 703
combo box · 751
comma operator · 152, 175
Command Pattern · 575
comments: and embedded documentation · 122
common interface · 325
common pitfalls when using operators · 153
Common-Gateway Interface (CGI) · 948
Comparable · 432, 475
Comparator · 434, 475
compareTo( ), in java.lang.Comparable · 432
comparing arrays · 431
compilation unit · 245
compile-time constant · 294
compiling a Java program · 121
component, and JavaBeans · 801
composition · 37, 271; and cloning · 1027; and design · 340; and dynamic behavior change · 341; choosing composition vs. inheritance · 288; combining composition & inheritance · 281; vs. inheritance · 294, 642
compression: compression library · 606
concept, high · 75
ConcurrentModificationException · 515
conditional operator · 151
conference, Software Development Conference · 10
Console: Swing display framework in com.bruceeckel.swing · 702
console input · 597
const, in C++ · 1053
constant: compile-time constant · 294; folding · 294; groups of constant values · 359; implicit constants, and String · 1053
constrained properties · 818
constructor · 191; and anonymous inner classes · 370; and exception handling · 562; and exceptions · 561; and finally · 562; and overloading · 194; and polymorphism · 330; arguments · 193; base-class constructor · 332; base-class constructors and exceptions · 281; behavior of polymorphic methods inside constructors · 337; C++ copy constructor · 1042; calling base-class constructors with arguments · 280; calling from other constructors · 205; default · 202; default constructors · 196; initialization during inheritance and composition · 281; name · 192; no-arg constructors · 196; order of constructor calls with inheritance · 330; return value · 193; static construction clause · 228; synthesized default constructor access · 681
Constructor: for reflection · 678

consulting & mentoring provided by Bruce Eckel · 23
container: class · 407, 439; of primitives · 412
container classes, utilities for · 444
continue keyword · 175
control: access · 36
control framework, and inner classes · 394
controlling access · 267
conversion: automatic · 273; narrowing conversion · 155, 201; widening conversion · 155
cookies: and JSP · 971
cookies, and servlets · 955
copy: deep copy · 1020; shallow copy · 1019
copying an array · 429
CORBA · 980
costs, startup · 95
coupling · 537
CRC, class-responsibility-collaboration cards · 79
CRC32 · 608
createStatement( ) · 930
critical section, and synchronized block · 852

D

daemon threads · 840
data: final · 294; primitive data types and use with operators · 159; static initialization · 225
data type: equivalence to class · 33
database: flat-file database · 932; Java DataBase Connectivity (JDBC) · 927; relational database · 933; URL · 928
DatabaseMetaData · 938
DataFlavor · 792
Datagram · 923; User Datagram Protocol (UDP) · 923
DataInput · 593
DataInputStream · 586, 591, 597, 599
DataOutput · 593
DataOutputStream · 588, 592, 599
dead, Thread · 859
deadlock, multithreading · 865, 872
death condition, and finalize( ) · 214
decorator design pattern · 585
decoupling: via polymorphism · 46
decoupling through polymorphism · 311
decrement operator · 139
deep copy · 1020, 1027; and ArrayList · 1030; using serialization to perform deep copying · 1032
default constructor · 196, 202; synthesizing a default constructor · 279
default constructor, access the same as the class · 681
default keyword, in a switch statement · 183
default package · 257
DefaultMutableTreeNode · 784
defaultReadObject( ) · 629
DefaultTreeModel · 784
defaultWriteObject( ) · 629
DeflaterOutputStream · 606
Delphi, from Borland · 800
Demarco, Tom · 1095
dequeue · 440
derived: derived class · 315; derived class, initializing · 278; types · 39
design · 342; adding more methods to a design · 268; analysis and design, object-oriented · 71; and composition · 340; and inheritance · 339; and mistakes · 268; five stages of object design · 82; library design · 243; of object hierarchies · 307; patterns · 86, 94
design patterns · 266; decorator · 585; singleton · 266
destroy( ) · 877
destructor · 208, 209, 554; Java doesn't have one · 283
development, incremental · 291
diagram: inheritance · 47; use case · 77
diagram, class inheritance diagrams · 293
dialog box · 771
dialog, file · 776
dialog, tabbed · 755
dictionary · 477
digital signing · 692
directory: and packages · 254; creating directories and paths · 578; lister · 574
display framework, for Swing · 702
dispose( ) · 772
division · 137
documentation: comments & embedded documentation · 122
Domain Name System (DNS) · 905
dotted quad · 905
double, literal value marker (D) · 156

do-while · 173
downcast · 293, 343, 666; type-safe downcast in run-time type identification · 665
Drawing lines in Swing · 768
drop-down list · 751
dynamic: behavior change with composition · 341; binding · 311, 316
dynamic aggregate initialization syntax for arrays · 412

E

early binding · 45, 315
East, BorderLayout · 713
editor, creating one using the Swing JTextPane · 747
efficiency: and arrays · 408; and final · 302; and threads · 828; when using the synchronized keyword · 853
EJB · 990
elegance, in programming · 87
else keyword · 171
encapsulation · 261
Enterprise JavaBeans (EJB) · 990
enum, groups of constant values in C & C++ · 359
Enumeration · 520
equals( ) · 142, 475; and hashed data structures · 485; overriding for HashMap · 484; vs. == · 645
equivalence: == · 141; object equivalence · 141
error: handling with exceptions · 531; recovery · 568; reporting errors in book · 23; standard error stream · 538
event: event-driven system · 394; JavaBeans · 801; multicast · 796; multicast event and JavaBeans · 854; responding to a Swing event · 707; Swing event model · 794; unicast · 796
event listener · 722; order of execution · 796
event model, Swing · 722
event-driven programming · 707
events and listeners · 723
EventSetDescriptors · 808
evolution, in program development · 85
exception: and base-class constructors · 281; and constructors · 561; and inheritance · 558, 566; catching an exception · 534; catching any exception · 543; changing the point of origin of the exception · 547; class hierarchies · 567; constructors · 562; creating your own · 537; design issues · 565; Error class · 549; Exception class · 549; *exception handler* · 535; exception handling · 531; exception matching · 566; FileNotFoundException · 565; fillInStackTrace( ) · 545; finally · 552; guarded region · 535; handler · 532; handling · 283; losing an exception, pitfall · 557; NullPointerException · 550; printStackTrace( ) · 545; restrictions · 558; re-throwing an exception · 545; RuntimeException · 550; specification · 542; termination vs. resumption · 536; Throwable · 543; throwing an exception · 533; try · 554; try block · 535; typical uses of exceptions · 568
exceptional condition · 532
exceptions: and JNI · 1074
executeQuery( ) · 930
Exponential notation · 156
extending a class during inheritance · 41
extends · 260, 277, 342; and interface · 359; keyword · 275
extensible: program · 320
extension: pure inheritance vs. extension · 341
extension, sign · 147
extension, zero · 147
Externalizable · 620; alternative approach to using · 626
Extreme Programming (XP) · 88, 1093

F

fail fast containers · 515
false · 143
FeatureDescriptor · 818
Field, for reflection · 678
fields, initializing fields in interfaces · 361
FIFO · 472
file: characteristics of files · 578; File.list( ) · 574; incomplete output files, errors and flushing · 599; JAR file · 245
File · 582, 592, 655; class · 574
file dialogs · 776
File Transfer Protocol (FTP) · 699
FileDescriptor · 582

FileInputReader · 597
FileInputStream · 582
FilenameFilter · 574, 653
FileNotFoundException · 565
FileOutputStream · 583
FileReader · 563, 590
FileWriter · 590, 599
fillInStackTrace( ) · 545
FilterInputStream · 582
FilterOutputStream · 583
FilterReader · 591
FilterWriter · 591
final · 350; and efficiency · 302; and private · 299; and static · 294; argument · 298, 577; blank finals · 297; classes · 301; data · 294; keyword · 294; method · 316; methods · 299, 339; static primitives · 296; with object references · 295
finalize( ) · 207, 566; and inheritance · 333; and super · 335; calling directly · 210; order of finalization of objects · 336
finally · 283, 286; and constructors · 562; keyword · 552; pitfall · 557
finding .class files during loading · 247
flat-file database · 932
flavor, clipboard · 790
float, literal value marker(F) · 156
floating point: true and false · 144
FlowLayout · 714
flushing output files · 599
focus traversal · 691
folding, constant · 294
for keyword · 173
forName( ) · 664, 727
FORTRAN · 156
forward referencing · 222
Fowler, Martin · 72, 85, 1093
framework: application framework and applets · 694; control framework and inner classes · 394
friendly · 243, 368; and interface · 350; and protected · 290; less accessible than protected · 335
FTP: File Transfer Protocol (FTP) · 699
function: member function · 35; overriding · 42
functor · 575

G

garbage collection · 207, 210, 333; and cleanup · 283; and native method execution · 1073; forcing finalization · 286; how the collector works · 215; order of object reclamation · 286; reachable objects · 495; setting references to null to allow cleanup · 397
generator · 443
generator object, to fill arrays and containers · 416
get( ), ArrayList · 450, 456
get( ), HashMap · 481
getBeanInfo( ) · 805
getBytes( ) · 598
getClass( ) · 544, 674
getConstructor( ) · 737
getConstructors( ) · 681
getContentPane( ) · 695
getContents( ) · 792
getEventSetDescriptors( ) · 808
getFloat( ) · 930
getInputStream( ) · 909
getInt( ) · 930
getInterfaces( ) · 676
getMethodDescriptors( ) · 808
getMethods( ) · 681
getModel( ) · 784
getName( ) · 677, 808
getOutputStream( ) · 909
getPriority( ) · 878
getPropertyDescriptors( ) · 808
getPropertyType( ) · 808
getReadMethod( ) · 808
getSelectedValues( ) · 753
getState( ) · 765
getString( ) · 930
getSuperclass( ) · 676
getTransferData( ) · 792
getTransferDataFlavors( ) · 792
getWriteMethod( ) · 808
Glass, Robert · 1094
glue, in BoxLayout · 717
goto: lack of goto in Java · 177
graphical user interface (GUI) · 394, 689
graphics · 776
Graphics · 768
greater than (>) · 141
greater than or equal to (>=) · 141
GridBagLayout · 716
GridLayout · 715, 894

guarded region, in exception handling · 535
GUI: graphical user interface · 394, 689
GUI builders · 690
guidelines: object development · 83
guidelines, coding standards · 1077
GZIPInputStream · 606
GZIPOutputStream · 606

H

handler, exception · 535
hardware devices, interfacing with · 1065
has-a · 37
has-a relationship, composition · 289
hash code · 477, 488
hash function · 488
hashCode( ) · 473, 477; and hashed data structures · 485; issues when writing · 492; overriding for HashMap · 484
hashing · 485; external chaining · 488; perfect hashing function · 488
HashMap · 476, 500, 733; used with ArrayList · 652
HashSet · 473, 506
Hashtable · 510, 521
hasNext( ), Iterator · 457
Hexadecimal · 156
hiding: implementation · 35
hiding, implementation · 261
high concept · 75
HTML · 948; name · 839; param · 839; value · 839
HTML on Swing components · 779

I

I/O: and threads, blocking · 860; available( ) · 598; blocking on I/O · 869; blocking, and available( ) · 598; BufferedInputStream · 586; BufferedOutputStream · 588; BufferedReader · 563, 591, 597; BufferedWriter · 591, 599; ByteArrayInputStream · 582; ByteArrayOutputStream · 583; characteristics of files · 578; CharArrayReader · 590; CharArrayWriter · 590; CheckedInputStream · 606; CheckedOutputStream · 606; close( ) · 597; compression library · 606; console input · 597; controlling the process of serialization · 619; DataInput · 593; DataInputStream · 586, 591, 597, 599; DataOutput · 593; DataOutputStream · 588, 592, 599; DeflaterOutputStream · 606; directory lister · 574; directory, creating directories and paths · 578; Externalizable · 620; File · 582, 592, 655; File class · 574; File.list( ) · 574; FileDescriptor · 582; FileInputReader · 597; FileInputStream · 582; FilenameFilter · 574, 653; FileOutputStream · 583; FileReader · 563, 590; FileWriter · 590, 599; FilterInputStream · 582; FilterOutputStream · 583; FilterReader · 591; FilterWriter · 591; from standard input · 602; GZIPInputStream · 606; GZIPOutputStream · 606; InflaterInputStream · 606; input · 581; InputStream · 581, 913; InputStreamReader · 589, 590, 913; internationalization · 590; library · 573; lightweight persistence · 613; LineNumberInputStream · 586; LineNumberReader · 591; mark( ) · 593; mkdirs( ) · 580; nextToken( ) · 654; ObjectOutputStream · 614; output · 581; OutputStream · 581, 583, 913; OutputStreamWriter · 589, 590, 913; pipe · 581; piped stream · 869; piped streams · 602; PipedInputStream · 582; PipedOutputStream · 582, 583; PipedReader · 590; PipedWriter · 590; PrintStream · 588; PrintWriter · 591, 599, 913; pushBack( ) · 654; PushbackInputStream · 586; PushBackReader · 591; RandomAccessFile · 592, 593, 599; read( ) · 581; readChar( ) · 600; readDouble( ) · 600; Reader · 581, 589, 590, 913; readExternal( ) · 620; readLine( ) · 565, 591, 599, 600, 603; readObject( ) · 614; redirecting standard I/O · 604; renameTo( ) · 580; reset( ) · 593; seek( ) · 593, 601; SequenceInputStream · 582, 592; Serializable · 620; setErr(PrintStream) · 604; setIn(InputStream) · 604;

setOut(PrintStream) · 604; StreamTokenizer · 591, 639, 653, 682; StringBuffer · 582; StringBufferInputStream · 582; StringReader · 590, 597; StringWriter · 590; System.err · 602; System.in · 597, 602; System.out · 602; transient · 624; typical I/O configurations · 594; Unicode · 590; write( ) · 581; writeBytes( ) · 600; writeChars( ) · 600; writeDouble( ) · 600; writeExternal( ) · 620; writeObject( ) · 614; Writer · 581, 589, 590, 913; ZipEntry · 610; ZipInputStream · 606; ZipOutputStream · 606
Icon · 738
IDL · 982
idltojava · 984
if-else statement · 151, 171
IllegalMonitorStateException · 866
ImageIcon · 738
immutable objects · 1047
implementation · 34; and interface · 288, 350; and interface, separating · 36; and interface, separation · 262; hiding · 35, 261, 368; separation of interface and implementation · 722
implements keyword · 350
import keyword · 244
increment operator · 139
incremental development · 291
indexed property · 818
indexing operator [ ] · 231
indexOf( ): String · 576, 681
InflaterInputStream · 606
inheritance · 38, 260, 271, 275, 311; and cloning · 1034; and final · 302; and finalize( ) · 333; and synchronized · 858; choosing composition vs. inheritance · 288; class inheritance diagrams · 293; combining composition & inheritance · 281; designing with inheritance · 339; diagram · 47; extending a class during · 41; extending interfaces with inheritance · 358; from an abstract class · 326; from inner classes · 384; inheritance and method overloading vs. overriding · 286; initialization with inheritance · 304; multiple inheritance in C++ and Java · 354; pure inheritance vs. extension · 341; specialization · 289; vs composition · 642; vs. composition · 294
initial capacity, of a HashMap or HashSet · 491
initialization: and class loading · 304; array initialization · 231; base class · 278; class member · 273; constructor initialization during inheritance and composition · 281; initializing class members at point of definition · 221; initializing with the constructor · 191; instance initialization · 229, 375; member initializers · 332; non-static instance initialization · 229; of class data members · 220; of method variables · 220; order of initialization · 223, 338; static · 306; with inheritance · 304
inizialization: lazy · 273
inline method calls · 299
inner class · 365, 799; access rights · 376; and super · 385; and overriding · 385; and control frameworks · 394; and Swing · 722; and upcasting · 368; anonymous · 709; anonymous inner class · 576, 875; anonymous inner class and constructors · 375; anonymous, and table-driven code · 502; callback · 391; closure · 391; hidden reference to the object of the enclosing class · 378; identifiers and .class files · 387; in methods & scopes · 370; inheriting from inner classes · 384; nesting within any arbitrary scope · 372; private · 833; private inner classes · 397; referring to the outer class object · 381; static inner classes · 379
input: console input · 597
InputStream · 581, 913
InputStreamReader · 589, 590, 913
insertNodeInto( ) · 784
instance: instance initialization · 375; non-static instance initialization · 229
instance of a class · 31
instanceof: dynamic instanceof · 671; keyword · 666
Integer: parseInt( ) · 776
Integer wrapper class · 233
interface: and implementation, separation · 262; and inheritance · 358; base-class interface · 320; Cloneable interface used as a flag · 1022; common

interface · 325; defining the class · 88; for an object · 32; graphical user interface (GUI) · 394, 689; implementation, separation of · 36; initializing fields in interfaces · 361; keyword · 349; nesting interfaces within classes and other interfaces · 362; private, as nested interfaces · 364; Runnable · 836; separation of interface and implementation · 722; upcasting to an interface · 353; user · 78; vs. abstract · 356; vs. implemenation · 288
Interface Definition Language (IDL) · 982
interfaces: name collisions when combining interfaces · 356
interfacing with hardware devices · 1065
internationalization, in I/O library · 590
Internet: Internet Protocol · 905; Internet Service Provider (ISP) · 699
interrupt( ) · 873
InterruptedException · 827
intranet · 693; and applets · 693
Introspector · 805
IP (Internet Protocol) · 905
is-a · 341; relationship, inheritance · 289; relationship, inheritance & upcasting · 292; vs. is-like-a relationships · 42
isDaemon( ) · 840
isDataFlavorSupported( ) · 792
isInstance · 671
isInterface( ) · 677
is-like-a · 342
ISP (Internet Service Provider) · 699
iteration, in program development · 84
iterator · 456
Iterator · 456, 463, 500; hasNext( ) · 457; next( ) · 457
iterator( ) · 463

J

Jacobsen, Ivar · 1093
JApplet · 713; menus · 759
JAR · 816; archive tag, for HTML and JAR files · 793; file · 245; jar files and classpath · 249; packaging applets to optimize loading · 793
JAR utility · 611
Java · 99; and pointers · 1013; and set-top boxes · 146; capitalization style source-code checking tool · 645; compiling and running a program · 121; containers library · 440; public Java seminars · 11; versions · 22
Java 1.1: I/O streams · 589
Java AWT · 689
Java Foundation Classes (JFC/Swing) · 689
Java operators · 133
Java programs, running from the Windows Explorer · 705
Java Server Pages (JSP) · 960
Java Virtual Machine · 662
JavaBeans: see Beans · 800
javac · 121
javah · 1067
JButton · 738
JButton, Swing · 706
JCheckBox · 738, 748
JCheckboxMenuItem · 765
JCheckBoxMenuItem · 760
JComboBox · 751
JComponent · 740, 768
JDBC: createStatement( ) · 930; database URL · 928; DatabaseMetaData · 938; executeQuery( ) · 930; flat-file database · 932; getFloat( ) · 930; getInt( ) · 930; getString( ) · 930; Java DataBase Connectivity · 927; join · 932; relational database · 933; ResultSet · 930; SQL stored procedures · 935; Statement · 930; Structured Query Language (SQL) · 927
JDialog · 771; menus · 759
JDK: downloading and installing · 121
JFC: Java Foundation Classes (JFC/Swing) · 689
JFileChooser · 776
JFrame · 704, 713; menus · 759
Jini · 1003
JIT: Just-In Time compilers · 98
JLabel · 695, 743
JList · 753
JMenu · 759, 765
JMenuBar · 759, 766
JMenuItem · 738, 759, 765, 766, 768
JNI functions · 1069
JNICALL · 1068
JNIEnv · 1069
JNIEXPORT · 1068
join · 932
JOptionPane · 756
JPanel · 713, 736, 768, 894

JPopupMenu · 766
JProgressBar · 781
JRadioButton · 738, 750
JScrollPane · 712, 744, 755, 784
JSlider · 781
JSP · 960
JTabbedPane · 755
JTable · 784
JTextArea · 711, 790
JTextField · 708, 740
JTextPane · 747
JToggleButton · 736
JTree · 781
JVM (Java Virtual Machine) · 662

K

keyboard navigation, and Swing · 691
keyboard shortcuts · 765
keySet( ) · 511
keywords: class · 32, 38
Koenig, Andrew · 1079

L

label · 178
labeled break · 178
labeled continue · 178
late binding · 45, 311, 316
layout: controlling layout with layout managers · 712
lazy inizialization · 273
left-shift operator (<<) · 147
length, array member · 232
length, for arrays · 409
less than (<) · 141
less than or equal to (<=) · 141
lexicographic vs. alphabetic sorting · 436
library: creator, vs. client programmer · 243; design · 243; use · 244
LIFO · 471
lightweight: Swing components · 691
lightweight persistence · 613
LineNumberInputStream · 586
LineNumberReader · 591
linked list · 440
LinkedList · 467, 472, 505
list: drop-down list · 751
List · 408, 439, 440, 467, 753; sorting and searching · 511
list boxes · 753
listener adapters · 729
listener classes · 799
listener interfaces · 728
listeners and events · 723
Lister, Timothy · 1095
ListIterator · 467
literal: class literal · 664, 669; double · 156; float · 156; long · 156; values · 155
load factor, of a HashMap or HashSet · 491
loading: .class files · 247; initialization & class loading · 304; loding a class · 305
local loopback IP address · 907
localhost · 907; and RMI · 977
lock, for multithreading · 848
logarithms: natural logarithms · 156
logical: AND · 154; operator and short-circuiting · 144; operators · 143; OR · 154
long, literal value marker (L) · 156
Look & Feel: Pluggable · 787
lvalue · 134

M

main( ) · 277
maintenance, program · 85
management obstacles · 95
manifest file, for JAR files · 611, 816
map · 477
Map · 408, 439, 440, 476, 508
Map.Entry · 486
mark( ) · 593
Math.random( ) · 480; values produced by · 186
mathematical operators · 137
max( ) · 512
member: member function · 35; object · 37
member initializers · 332
memory exhaustion, solution via References · 495
mentoring: and training · 95, 96
menu: JPopupMenu · 766
menus: JDialog, JApplet, JFrame · 759
message box, in Swing · 756
message, sending · 33
meta-class · 662
method: adding more methods to a design · 268; aliasing during a method

call · 1014; aliasing during method calls · 136; behavior of polymorphic methods inside constructors · 337; distinguishing overloaded methods · 196; final · 316, 339; final methods · 299; initialization of method variables · 220; inline method calls · 299; inner classes in methods & scopes · 370; lookup tool · 724; method call binding · 315; overloading · 194; passing a reference into a method · 1014; polymorphic method call · 311; private · 339; protected methods · 290; recursive · 459; static · 206; synchronized method and blocking · 860
Method · 808; for reflection · 678
MethodDescriptors · 808
methodology, analysis and design · 71
Meyers, Scott · 35
Microsoft · 820; Visual Basic · 800
min( ) · 512
mission statement · 75
mistakes, and design · 268
mkdirs( ) · 580
mnemonics (keyboard shortcuts) · 765
modulus · 137
monitor, for multithreading · 848
multicast · 814; event, and JavaBeans · 854; multicast events · 796
multidimensional arrays · 236
Multimedia CD ROM for book · 20
multiparadigm programming · 31
multiple inheritance, in C++ and Java · 354
multiplication · 137
multiply-nested class · 383
MultiStringMap · 652
multitasking · 825
multithreading · 825, 917; and containers · 514; and JavaBeans · 854; blocking · 859; deadlock · 865; deciding what methods to synchronize · 858; drawbacks · 899; Runnable · 891; servlets · 954; when to use it · 899
multi-tiered systems · 796

N

name · 697; clash · 244; collisions · 250; creating unique package names · 247; spaces · 244
name collisions when combining interfaces · 356
name, HTML keyword · 839
Naming: bind( ) · 976; rebind( ) · 978; unbind( ) · 978
narrowing conversion · 155, 170, 201
native method interface (NMI) in Java 1.0 · 1065
natural logarithms · 156
nesting interfaces · 362
network programming · 904; accept( ) · 909; client · 907; Common-Gateway Interface (CGI) · 948; datagrams · 923; dedicated connection · 917; displaying a Web page from within an applet · 923; DNS (Domain Name System) · 905; dotted quad · 905; getInputStream( ) · 909; getOutputStream( ) · 909; HTML · 948; identifying machines · 905; Internet Protocol (IP) · 905; Java DataBase Connectivity (JDBC) · 927; local loopback IP address · 907; localhost · 907; multithreading · 917; port · 908; reliable protocol · 923; server · 907; serving multiple clients · 917; showDocument( ) · 924; Socket · 915; stream-based sockets · 923; testing programs without a network · 907; Transmission Control Protocol (TCP) · 923; unreliable protocol · 923; URL · 925; User Datagram Protocol (UDP) · 923
new operator · 207; and primitives, array · 233
newInstance( ) · 737; reflection · 676
next( ), Iterator · 457
nextToken( ) · 654
NMI: Java 1.0 native method interface · 1065
no-arg: constructors · 196
non-Java code, calling · 1065
North, BorderLayout · 713
NOT: logical (!) · 143
not equivalent (!=) · 141
notify( ) · 860
notifyAll( ) · 860
notifyListeners( ) · 858
null · 107, 411; garbage collection, allowing cleanup · 397
NullPointerException · 550
numbers, binary · 156

O

object · 31; aliasing · 136; arrays are first-class objects · 409; assigning objects by copying references · 134; assignment and reference copying · 134; business object/logic · 796; Class object · 633, 662, 848; creation · 192; equals( ) method · 142; equivalence · 141; equivalence vs reference equivalence · 142; final · 295; five stages of object design · 82; guidelines for object development · 83; immutable objects · 1047; interface to · 32; lock, for multithreading · 848; member · 37; object-oriented programming · 660; order of finalization of objects · 336; process of creation · 227; reference equivalence vs. object equivalence · 1025; serialization · 613; web of objects · 614, 1020
Object · 408; clone( ) · 1021, 1025; getClass( ) · 674; hashCode( ) · 477; standard root class, default inheritance from · 275; wait( ) and notify( ) methods · 866
object-oriented: analysis and design · 71; basic concepts of object-oriented programming (OOP) · 29
ObjectOutputStream · 614
obstacles, management · 95
Octal · 156
ODBC · 928
OMG · 981
ones complement operator · 146
OOP · 262; analysis and design · 71; basic characteristics · 31; basic concepts of object-oriented programming · 29; protocol · 350; Simula programming language · 32; substitutability · 31
operator · 133; + and += overloading for String · 277; +, for String · 1054; == and != · 1025; binary · 146; bitwise · 146; casting · 154; comma · 152; comma operator · 175; common pitfalls · 153; indexing operator [ ] · 231; logical · 143; logical operators and short-circuiting · 144; ones-complement · 146; operator overloading for String · 1054; overloading · 153; precedence · 134; precedence mnemonic · 158; relational · 141; shift · 147; ternary · 151; unary · 139, 146
optional methods, in the Java 2 containers · 516
OR · 154; (||) · 143
order: of constructor calls with inheritance · 330; of finalization of objects · 336; of initialization · 223, 304, 338
organization, code · 255
OutputStream · 581, 583, 913
OutputStreamWriter · 589, 590, 913
overflow: and primitive types · 169
overloading: and constructors · 194; distinguishing overloaded methods · 196; lack of name hiding during inheritance · 286; method overloading · 194; on return values · 202; operator + and += overloading for String · 277; operator overloading · 153; operator overloading for String · 1054; overloading vs. overriding · 286; vs. overriding · 324
overriding: and inner classes · 385; function · 42; overloading vs. overriding · 286; vs. overloading · 324

P

package · 244; access, and friendly · 255; and applets · 699; and directory structure · 254; creating unique package names · 247; default package · 257; names, capitalization · 116; visibility, friendly · 368
paintComponent( ) · 768, 776
Painting on a JPanel in Swing · 768
pair programming · 90
paralysis, analysis · 72
param, HTML keyword · 839
parameter, applet · 697
parameterized type · 455
parseInt( ) · 776
pass: pass by value · 1018; passing a reference into a method · 1014
Pattern: Command Pattern · 575
patterns, design · 86, 94
patterns, design patterns · 266
perfect hashing function · 488
performance: and final · 302
performance issues · 96
Perl programming language · 705

persistence · 630; lightweight persistence · 613
PhantomReference · 495
pipe · 581
piped stream · 869
piped streams · 602
PipedInputStream · 582
PipedOutputStream · 582, 583
PipedReader · 590
PipedWriter · 590
planning, software development · 74
Plauger, P.J. · 1094
Pluggable Look & Feel · 787
pointer: Java exclusion of pointers · 391
pointers, and Java · 1013
polymorphism · 44, 311, 346, 660, 685; and constructors · 330; behavior of polymorphic methods inside constructors · 337
port · 908
portability in C, C++ and Java · 158
position, absolute, when laying out Swing components · 716
precedence: operator precedence mnemonic · 158
prerequisites, for this book · 29
primitive: comparison · 142; containers of primitives · 412; data types, and use with operators · 159; dealing with the immutability of primitive wrapper classes · 1047; final · 294; final static primitives · 296; initialization of class data members · 220; wrappers · 481
primitive types · 105
printInfo( ) · 677
printing arrays · 417
println( ) · 458
printStackTrace( ) · 543, 545
PrintStream · 588
PrintWriter · 591, 599, 913
priority: default priority for a Thread group · 882; thread · 877
private · 36, 243, 255, 258, 290, 848; illusion of overriding private methods · 299; inner class · 833; inner classes · 397; interfaces, when nested · 364; methods · 339
problem space · 30, 291
process, and threading · 825
program: maintenance · 85
programmer, client · 35
programming: basic concepts of object-oriented programming (OOP) · 29; coding standards · 1077; event-driven programming · 707; Extreme Programming (XP) · 88, 1093; in the large · 92; multiparadigm · 31; object-oriented · 660; pair · 90
progress bar · 780
promotion: of primitive types · 169; type promotion · 157
Properties · 652
property · 801; bound properties · 818; constrained properties · 818; custom property editor · 818; custom property sheet · 818; indexed property · 818
PropertyChangeEvent · 818
PropertyDescriptors · 808
ProptertyVetoException · 818
protected · 36, 243, 255, 260, 290; and friendly · 290; is also friendly · 261; more accessible than friendly · 335; use in clone( ) · 1021
protocol · 350; unreliable protocol · 923
prototyping: rapid · 86
public · 36, 243, 255, 256; and interface · 350; class, and compilation units · 245
pure: substitution · 42
pure inheritance, vs. extension · 341
pure substitution · 342
pushBack( ) · 654
PushbackInputStream · 586
PushBackReader · 591
put( ), HashMap · 481
Python · 81, 99

Q

queue · 440, 472

R

RAD (Rapid Application Development) · 678
radio button · 750
random number generator, values produced by · 186
random( ) · 480
RandomAccessFile · 592, 593, 599
rapid prototyping · 86

reachable objects and garbage collection · 495
read( ) · 581
readChar( ) · 600
readDouble( ) · 600
Reader · 581, 589, 590, 869, 913
readExternal( ) · 620
reading from standard input · 602
readLine( ) · 565, 591, 599, 600, 603
readObject( ) · 614; with Serializable · 627
rebind( ) · 978
recursion, unintended via toString() · 459
redirecting standard I/O · 604
refactoring · 85
reference: assigning objects by copying references · 134; equivalence vs object equivalence · 142; final · 295; finding exact type of a base reference · 662; null · 107; reference equivalence vs. object equivalence · 1025
Reference, from java.lang.ref · 495
referencing, forward referencing · 222
reflection · 677, 678, 724, 804; and Beans · 801; difference between RTTI and reflection · 679
reflection example · 736
registry: remote object registry · 976
relational: database · 933; operators · 141
reliable protocol · 923
Remote Method Invocation (RMI) · 973
RemoteException · 980
removeActionListener( ) · 812, 857
removeXXXListener( ) · 723
renameTo( ) · 580
reporting errors in book · 23
request, in OOP · 33
requirements analysis · 75
reset( ) · 593
ResultSet · 930
resume( ) · 860, 864; and deadlocks · 873; deprecation in Java 2 · 875
resumption, termination vs. resumption, exception handling · 536
re-throwing an exception · 545
return: an array · 413; constructor return value · 193; overloading on return value · 202
reusability · 37
reuse · 83; code reuse · 271; existing class libraries · 94; reusable code · 800
right-shift operator (>>) · 147
RMI: AlreadyBoundException · 978; and CORBA · 989; bind( ) · 976; localhost · 977; rebind( ) · 978; Remote · 974; remote interface · 974; Remote Method Invocation · 973; remote object registry · 976; RemoteException · 974, 980; rmic · 979; rmic and classpath · 979; rmiregistry · 976; RMISecurityManager · 976; Serializable arguments · 978; skeleton · 978; stub · 978; TCP/IP · 977; unbind( ) · 978; UnicastRemoteObject · 974
rmic · 979
rmiregistry · 976
RMISecurityManager · 976
rollover · 740
RTTI: and cloning · 1025; cast · 661; Class · 737; Class object · 662; ClassCastException · 666; Constructor · 678; difference between RTTI and reflection · 679; downcast · 666; Field · 678; getConstructor( ) · 737; instanceof keyword · 666; isInstance · 671; meta-class · 662; Method · 678; newInstance( ) · 737; reflection · 677; run-time type identification (RTTI) · 344; type-safe downcast · 665; using the Class object · 674
Rumbaugh, James · 1093
runFinalizersOnExit( ) · 335
Runnable · 891; interface · 836; Thread · 859
running a Java program · 121
run-time binding · 316; polymorphism · 311
run-time type identification: (RTTI) · 344; misuse · 685; shape example · 659; when to use it · 685
RuntimeException · 408, 550
rvalue · 134

S

safety, and applet restrictions · 692
scenario · 77
scheduling · 79
scope: inner class nesting within any arbitrary scope · 372; inner classes in methods & scopes · 370; use case · 84
scrolling in Swing · 712
searching: sorting and searching Lists · 511

searching an array · 437
section, critical section and synchronized block · 852
seek( ) · 593, 601
seminars: public Java seminars · 11; training, provided by Bruce Eckel · 23
sending a message · 33
separating business logic from UI logic · 796
separation of interface and implementation · 36, 262, 722
SequenceInputStream · 582, 592
Serializable · 613, 620, 625, 637, 814; readObject( ) · 627; writeObject( ) · 627
serialization: and object storage · 630; and transient · 624; controlling the process of serialization · 619; defaultReadObject( ) · 629; defaultWriteObject( ) · 629; RMI arguments · 978; to perform deep copying · 1032; Versioning · 630
server · 907
servlet · 948; multithreading · 954; running servlets with Tomcat · 960; session tracking · 955
session: and JSP · 969
session tracking, with servlets · 955
Set · 408, 439, 440, 473, 506
setActionCommand( ) · 765
setBorder( ) · 743
setContents( ) · 792
setDaemon( ) · 840
setDefaultCloseOperation( ) · 704
setErr(PrintStream) · 604
setIcon( ) · 740
setIn(InputStream) · 604
setLayout( ) · 713
setMnemonic( ) · 765
setOut(PrintStream) · 604
setPriority( ) · 878
setToolTipText( ) · 740
shallow copy · 1019, 1027
shape: example · 39, 316; example, and run-time type identification · 659
shift operators · 147
short-circuit, and logical operators · 144
shortcut, keyboard · 765
show( ) · 773
showDocument( ) · 924
shuffle( ) · 512
side effect · 133, 141, 202, 1016
sign extension · 147
signed two's complement · 151
Simula programming language · 32
Simula-67 · 262
sine wave · 768
singleton: design pattern · 266
size( ), ArrayList · 451
Size, of a HashMap or HashSet · 491
sizeof( ): lack of in Java · 158
skeleton, RMI · 978
sleep( ) · 827, 846, 860, 862
slider · 780
Smalltalk · 31, 207
Socket · 915
sockets, stream-based · 923
SoftReference · 495
software: development methodology · 72
Software Development Conference · 10
solution space · 30
sorting · 431; and searching Lists · 511
source code copyright notice · 20
South, BorderLayout · 713
space: problem · 30; solution · 30
specialization · 289
specification: system specification · 75
specification, exception · 542
specifier: access specifiers · 36, 243, 255
SQL: stored procedures · 935; Structured Query Language · 927
Stack · 471, 521
standard input: Reading from standard input · 602
standards: coding standards · 22, 1077
startup costs · 95
stateChanged( ) · 771
statement: mission · 75
Statement · 930
static · 350; and final · 294; block · 228; clause · 664; construction clause · 228; data initialization · 225; final static primitives · 296; initialization · 306; inner classes · 379; keyword · 206; method · 206; synchronized static · 848
STL: C++ · 440
stop( ): and deadlocks · 873; deprecation in Java 2 · 873
stored procedures in SQL · 935
stream, I/O · 581
stream-based sockets · 923
StreamTokenizer · 591, 639, 653, 682
String: automatic type conversion · 454; class methods · 1052; concatenation with operator + · 153; immutability ·

1052; indexOf( ) · 576, 681; lexicographic vs. alphabetic sorting · 436; methods · 1056; operator + · 454; Operator + · 153; operator + and += overloading · 277; toString( ) · 272, 452
StringBuffer · 582; methods · 1058
StringBufferInputStream · 582
StringReader · 590, 597
StringSelection · 792
StringTokenizer · 642
StringWriter · 590
struts, in BoxLayout · 717
stub, RMI · 978
style of creating classes · 262
subobject · 278, 288
substitutability, in OOP · 31
substitution: principle · 42
subtraction · 137
super · 280; and finalize( ) · 335; and inner classes · 385
super keyword · 278
super.clone( ) · 1021, 1025, 1041
superclass · 278
suspend( ) · 860, 864; and deadlocks · 873; deprecation in Java 2 · 875
Swing · 689
Swing component examples · 734
Swing components, using HTML with · 779
Swing event model · 722, 794
switch keyword · 183
synchronized · 59, 848; and inheritance · 858; and wait( ) & notify( ) · 866; containers · 514; deciding what methods to synchronize · 858; efficiency · 853; method, and blocking · 860; static · 848; synchronized block · 852
system clipboard · 790
system specification · 75
System.arraycopy( ) · 429
System.err · 538, 602
System.gc( ) · 213
System.in · 597, 602
System.out · 602
System.out.println( ) · 458
System.runFinalization( ) · 213

T

tabbed dialog · 755
table · 784
table-driven code, and anonymous inner classes · 502
TCP, Transmission Control Protocol · 923
TCP/IP, and RMI · 977
template: in C++ · 455
termination vs. resumption, exception handling · 536
ternary operator · 151
testing: automated · 89; Extreme Programming (XP) · 88; unit testing · 277
testing techniques · 381
this keyword · 203
Thread · 825, 827; and JavaBeans · 854; and Runnable · 891; blocked · 859; combined with main class · 834; daemon threads · 840; dead · 859; deadlock · 872; deciding what methods to synchronize · 858; destroy( ) · 877; drawbacks · 899; getPriority( ) · 878; I/O and threads, blocking · 860; interrupt( ) · 873; isDaemon( ) · 840; new Thread · 859; notify( ) · 860; notifyAll( ) · 860; order of execution of threads · 831; priority · 877; properly suspending & resuming · 874; resume( ) · 860, 864; resume( ) , deprecation in Java 2 · 875; resume( ), and deadlocks · 873; run( ) · 829; Runnable · 859; Runnable interface · 836; setDaemon( ) · 840; setPriority( ) · 878; sharing limited resources · 842; sleep( ) · 846, 860, 862; start( ) · 830; states · 859; stop( ) , deprecation in Java 2 · 873; stop( ), and deadlocks · 873; stopping · 873; suspend( ) · 860, 864; suspend( ) , deprecation in Java 2 · 875; suspend( ), and deadlocks · 873; synchronized method and blocking · 860; thread group · 882; thread group, default priority · 882; threads and efficiency · 828; wait( ) · 860, 866; when they can be suspended · 847; when to use threads · 899; yield( ) · 860
throw keyword · 534
Throwable · 547; base class for Exception · 543
throwing an exception · 533
time-critical code sections · 1065
toArray( ) · 511
token · 639
Tokenizing · 639

Tomcat, standard servlet container · 960
tool tips · 740
TooManyListenersException · 796, 814
toString( ) · 272, 452, 458, 500
training · 93; and mentoring · 95, 96
training seminars provided by Bruce Eckel · 23
Transferable · 792
transient keyword · 624
translation unit · 245
Transmission Control Protocol (TCP) · 923
tree · 781
TreeMap · 476, 510, 642
TreeSet · 473, 506
true · 143
try · 286, 554; try block in exceptions · 535
two's complement, signed · 151
type: base · 39; data type equivalence to class · 33; derived · 39; finding exact type of a base reference · 662; parameterized type · 455; primitive · 105; primitive data types and use with operators · 159; type checking and arrays · 408; type safety in Java · 154; type-safe downcast in run-time type identification · 665; weak typing · 45
TYPE field, for primitive class literals · 665
type safe sets of constants · 361
type-conscious ArrayList · 454

U

UDP, User Datagram Protocol · 923
UML · 81; indicating composition · 37; Unified Modeling Language · 35, 1093
unary: minus (-) · 139; operator · 146; operators · 139; plus (+) · 139
unbind( ) · 978
unicast · 814; unicast events · 796
UnicastRemoteObject · 974
Unicode · 590
Unified Modeling Language (UML) · 35, 1093
unit testing · 277
unmodifiable, making a Collection or Map unmodifiable · 513
unsupported methods, in the Java 2 containers · 516
UnsupportedOperationException · 516
upcasting · 47, 291, 312, 660; and interface · 353; inner classes and upcasting · 368
updates of the book · 22
URL · 925
use case · 76; iteration · 84; scope · 84
User Datagram Protocol (UDP) · 923
user interface · 78; and threads, for responsiveness · 831; responsive, with threading · 826

V

value: preventing change at run-time · 294
value, HTML keyword · 839
variable: defining a variable · 174; initialization of method variables · 220; variable argument lists (unknown quantity and type of arguments) · 235
vector: of change · 86
Vector · 505, 519, 521
vector of change · 397
versioning, serialization · 630
versions of Java · 22
visibility, package visibility, (friendly) · 368
visual: programming · 800
Visual Basic, Microsoft · 800
visual programming environments · 690

W

wait( ) · 860, 866
Waldrop, M. Mitchell · 1095
weak: weakly typed language · 45
WeakHashMap · 498
WeakReference · 495
Web: displaying a Web page from within an applet · 923; placing an applet inside a Web page · 695; safety, and applet restrictions · 692
web of objects · 614, 1020
West, BorderLayout · 713
while · 172
widening conversion · 155
wild-card · 73
WindowAdapter · 704
windowClosing( ) · 704, 771
windowed applications · 700

Windows Explorer, running Java programs from · 705
wrapper, dealing with the immutability of primitive wrapper classes · 1047
write( ) · 581
writeBytes( ) · 600
writeChars( ) · 600
writeDouble( ) · 600
writeExternal( ) · 620
writeObject( ) · 614; with Serializable · 627
Writer · 581, 589, 590, 869, 913

X

XOR · 146
XP, Extreme Programming · 88

Y

yield( ) · 860

Z

zero extension · 147
ZipEntry · 610
ZipInputStream · 606
ZipOutputStream · 606

Bruce Eckel's

HANDS-ON

JAVA™

SEMINAR

Learn the programming language of the World Wide Web

In this *step-by-step introduction* each carefully-chosen subject is covered in a lecture followed by hands-on programming exercises.

This course is for you if you can follow basic code examples written in C language syntax.

WWW.BRUCEECKEL.COM

Check www.BruceEckel.com
for in-depth details
and the date and location
of the next
Hands-On Java Seminar

- Based on this book
- Taught by Bruce Eckel
- Personal attention from Bruce Eckel and his seminar assistants
- Includes in-class programming exercises
- Intermediate/Advanced seminars also offered
- Hundreds have already enjoyed this seminar—

Bruce Eckel's Hands-On Java Seminar
Multimedia CD
It's like coming to the seminar!
Available at *www.BruceEckel.com*

- The *Hands-On Java Seminar* captured on a Multimedia CD!
- Overhead slides and synchronized audio voice narration for all the lectures. Just play it to see and hear the lectures!
- Created and narrated by Bruce Eckel.
- Based on the material in this book.
- Demo lecture available at *www.BruceEckel.com*

End-User License Agreement for Microsoft Software

IMPORTANT-READ CAREFULLY: This Microsoft End-User License Agreement ("EULA") is a legal agreement between you (either an individual or a single entity) and Microsoft Corporation for the Microsoft software product included in this package, which includes computer software and may include associated media, printed materials, and "online" or electronic documentation ("SOFTWARE PRODUCT"). The SOFTWARE PRODUCT also includes any updates and supplements to the original SOFTWARE PRODUCT provided to you by Microsoft. By installing, copying, downloading, accessing or otherwise using the SOFTWARE PRODUCT, you agree to be bound by the terms of this EULA. If you do not agree to the terms of this EULA, do not install, copy, or otherwise use the SOFTWARE PRODUCT.

SOFTWARE PRODUCT LICENSE

The SOFTWARE PRODUCT is protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. The SOFTWARE PRODUCT is licensed, not sold.
1. GRANT OF LICENSE. This EULA grants you the following rights:
1.1 License Grant. Microsoft grants to you as an individual, a personal nonexclusive license to make and use copies of the SOFTWARE PRODUCT for the sole purposes of evaluating and learning how to use the SOFTWARE PRODUCT, as may be instructed in accompanying publications or documentation. You may install the software on an unlimited number of computers provided that you are the only individual using the SOFTWARE PRODUCT.
1.2 Academic Use. You must be a "Qualified Educational User" to use the SOFTWARE PRODUCT in the manner described in this section. To determine whether you are a Qualified Educational User, please contact the Microsoft Sales Information Center/One Microsoft Way/Redmond, WA 98052-6399 or the Microsoft subsidiary serving your country. If you are a Qualified Educational User, you may either:
(i) exercise the rights granted in Section 1.1, OR
(ii) if you intend to use the SOFTWARE PRODUCT solely for instructional purposes in connection with a class or other educational program, this EULA grants you the following alternative license models:
(A) Per Computer Model. For every valid license you have acquired for the SOFTWARE PRODUCT, you may install a single copy of the SOFTWARE PRODUCT on a single computer for access and use by an unlimited number of

student end users at your educational institution, provided that all such end users comply with all other terms of this EULA, OR

(B) Per License Model. If you have multiple licenses for the SOFTWARE PRODUCT, then at any time you may have as many copies of the SOFTWARE PRODUCT in use as you have licenses, provided that such use is limited to student or faculty end users at your educational institution and provided that all such end users comply with all other terms of this EULA. For purposes of this subsection, the SOFTWARE PRODUCT is "in use" on a computer when it is loaded into the temporary memory (i.e., RAM) or installed into the permanent memory (e.g., hard disk, CD ROM, or other storage device) of that computer, except that a copy installed on a network server for the sole purpose of distribution to other computers is not "in use". If the anticipated number of users of the SOFTWARE PRODUCT will exceed the number of applicable licenses, then you must have a reasonable mechanism or process in place to ensure that the number of persons using the SOFTWARE PRODUCT concurrently does not exceed the number of licenses.

2. DESCRIPTION OF OTHER RIGHTS AND LIMITATIONS.

- Limitations on Reverse Engineering, Decompilation, and Disassembly. You may not reverse engineer, decompile, or disassemble the SOFTWARE PRODUCT, except and only to the extent that such activity is expressly permitted by applicable law notwithstanding this limitation.
- Separation of Components. The SOFTWARE PRODUCT is licensed as a single product. Its component parts may not be separated for use on more than one computer.
- Rental. You may not rent, lease or lend the SOFTWARE PRODUCT.
- Trademarks. This EULA does not grant you any rights in connection with any trademarks or service marks of Microsoft.
- Software Transfer. The initial user of the SOFTWARE PRODUCT may make a one-time permanent transfer of this EULA and SOFTWARE PRODUCT only directly to an end user. This transfer must include all of the SOFTWARE PRODUCT (including all component parts, the media and printed materials, any upgrades, this EULA, and, if applicable, the Certificate of Authenticity). Such transfer may not be by way of consignment or any other indirect transfer. The transferee of such one-time transfer must agree to comply with the terms of this EULA, including the obligation not to further transfer this EULA and SOFTWARE PRODUCT.
- No Support. Microsoft shall have no obligation to provide any product support for the SOFTWARE PRODUCT.
- Termination. Without prejudice to any other rights, Microsoft may terminate this EULA if you fail to comply with the terms and conditions of this EULA. In

such event, you must destroy all copies of the SOFTWARE PRODUCT and all of its component parts.

3. COPYRIGHT. All title and intellectual property rights in and to the SOFTWARE PRODUCT (including but not limited to any images, photographs, animations, video, audio, music, text, and "applets" incorporated into the SOFTWARE PRODUCT), the accompanying printed materials, and any copies of the SOFTWARE PRODUCT are owned by Microsoft or its suppliers. All title and intellectual property rights in and to the content which may be accessed through use of the SOFTWARE PRODUCT is the property of the respective content owner and may be protected by applicable copyright or other intellectual property laws and treaties. This EULA grants you no rights to use such content. All rights not expressly granted are reserved by Microsoft.

4. BACKUP COPY. After installation of one copy of the SOFTWARE PRODUCT pursuant to this EULA, you may keep the original media on which the SOFTWARE PRODUCT was provided by Microsoft solely for backup or archival purposes. If the original media is required to use the SOFTWARE PRODUCT on the COMPUTER, you may make one copy of the SOFTWARE PRODUCT solely for backup or archival purposes. Except as expressly provided in this EULA, you may not otherwise make copies of the SOFTWARE PRODUCT or the printed materials accompanying the SOFTWARE PRODUCT.

5. U.S. GOVERNMENT RESTRICTED RIGHTS. The SOFTWARE PRODUCT and documentation are provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable. Manufacturer is Microsoft Corporation/One Microsoft Way/Redmond, WA 98052-6399.

6. EXPORT RESTRICTIONS. You agree that you will not export or re-export the SOFTWARE PRODUCT, any part thereof, or any process or service that is the direct product of the SOFTWARE PRODUCT (the foregoing collectively referred to as the "Restricted Components"), to any country, person, entity or end user subject to U.S. export restrictions. You specifically agree not to export or re-export any of the Restricted Components (i) to any country to which the U.S. has embargoed or restricted the export of goods or services, which currently include, but are not necessarily limited to Cuba, Iran, Iraq, Libya, North Korea, Sudan and Syria, or to any national of any such country, wherever located, who intends to transmit or transport the Restricted Components back to such country; (ii) to any end-user who you know or have reason to know will utilize the Restricted Components in the design, development or production of nuclear, chemical or biological weapons; or (iii) to any end-user who has been

prohibited from participating in U.S. export transactions by any federal agency of the U.S. government. You warrant and represent that neither the BXA nor any other U.S. federal agency has suspended, revoked, or denied your export privileges.
7. NOTE ON JAVA SUPPORT. THE SOFTWARE PRODUCT MAY CONTAIN SUPPORT FOR PROGRAMS WRITTEN IN JAVA. JAVA TECHNOLOGY IS NOT FAULT TOLERANT AND IS NOT DESIGNED, MANUFACTURED, OR INTENDED FOR USE OR RESALE AS ON-LINE CONTROL EQUIPMENT IN HAZARDOUS ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS IN THE OPERATION OF NUCLEAR FACILITIES, AIRCRAFT NAVIGATION OR COMMUNICATION SYSTEMS, AIR TRAFFIC CONTROL, DIRECT LIFE SUPPORT MACHINES, OR WEAPONS SYSTEMS, IN WHICH THE FAILURE OF JAVA TECHNOLOGY COULD LEAD DIRECTLY TO DEATH, PERSONAL INJURY, OR SEVERE PHYSICAL OR ENVIRONMENTAL DAMAGE.

MISCELLANEOUS

If you acquired this product in the United States, this EULA is governed by the laws of the State of Washington.
If you acquired this product in Canada, this EULA is governed by the laws of the Province of Ontario, Canada. Each of the parties hereto irrevocably attorns to the jurisdiction of the courts of the Province of Ontario and further agrees to commence any litigation which may arise hereunder in the courts located in the Judicial District of York, Province of Ontario.
If this product was acquired outside the United States, then local law may apply.
Should you have any questions concerning this EULA, or if you desire to contact Microsoft for any reason, please contact
Microsoft, or write: Microsoft Sales Information Center/One Microsoft Way/Redmond, WA 98052-6399.

LIMITED WARRANTY

LIMITED WARRANTY. Microsoft warrants that (a) the SOFTWARE PRODUCT will perform substantially in accordance with the accompanying written materials for a period of ninety (90) days from the date of receipt, and (b) any Support Services provided by Microsoft shall be substantially as described in applicable written materials provided to you by Microsoft, and Microsoft support engineers will make commercially reasonable efforts to solve any problem. To the extent allowed by applicable law, implied warranties on the SOFTWARE PRODUCT, if any, are limited to ninety (90) days. Some states/jurisdictions do

not allow limitations on duration of an implied warranty, so the above limitation may not apply to you.
CUSTOMER REMEDIES. Microsoft's and its suppliers' entire liability and your exclusive remedy shall be, at Microsoft's option, either (a) return of the price paid, if any, or (b) repair or replacement of the SOFTWARE PRODUCT that does not meet Microsoft's Limited Warranty and that is returned to Microsoft with a copy of your receipt. This Limited Warranty is void if failure of the SOFTWARE PRODUCT has resulted from accident, abuse, or misapplication. Any replacement SOFTWARE PRODUCT will be warranted for the remainder of the original warranty period or thirty (30) days, whichever is longer. Outside the United States, neither these remedies nor any product support services offered by Microsoft are available without proof of purchase from an authorized international source.
NO OTHER WARRANTIES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, MICROSOFT AND ITS SUPPLIERS DISCLAIM ALL OTHER WARRANTIES AND CONDITIONS, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, WITH REGARD TO THE SOFTWARE PRODUCT, AND THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES. THIS LIMITED WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY HAVE OTHERS, WHICH VARY FROM STATE/JURISDICTION TO STATE/JURISDICTION.
LIMITATION OF LIABILITY. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL MICROSOFT OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE PRODUCT OR THE FAILURE TO PROVIDE SUPPORT SERVICES, EVEN IF MICROSOFT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN ANY CASE, MICROSOFT'S ENTIRE LIABILITY UNDER ANY PROVISION OF THIS EULA SHALL BE LIMITED TO THE GREATER OF THE AMOUNT ACTUALLY PAID BY YOU FOR THE SOFTWARE PRODUCT OR U.S.$5.00; PROVIDED, HOWEVER, IF YOU HAVE ENTERED INTO A MICROSOFT SUPPORT SERVICES AGREEMENT, MICROSOFT'S ENTIRE LIABILITY REGARDING SUPPORT SERVICES SHALL BE GOVERNED BY THE TERMS OF THAT AGREEMENT. BECAUSE SOME STATES/JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.
0495 Part No. 64358

LICENSE AGREEMENT FOR MindView, Inc.'s
Thinking in C: Foundations for Java & C++ CD ROM
by Chuck Allison
This CD is provided together with the book "Thinking in Java, 2nd edition."

READ THIS AGREEMENT BEFORE USING THIS "Thinking in C: Foundations for C++ & Java" (Hereafter called "CD"). BY USING THE CD YOU AGREE TO BE BOUND BY THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS AND CONDITIONS OF THIS AGREEMENT, IMMEDIATELY RETURN THE UNUSED CD FOR A FULL REFUND OF MONIES PAID, IF ANY.

©2000 MindView, Inc. All rights reserved. Printed in the U.S.

SOFTWARE REQUIREMENTS
The purpose of this CD is to provide the Content, not the associated software necessary to view the Content. The Content of this CD is in HTML for viewing with Microsoft Internet Explorer 4 or newer, and uses Microsoft Sound Codecs available in Microsoft's Windows Media Player for Windows. It is your responsibility to correctly install the appropriate Microsoft software for your system.

The text, images, and other media included on this CD ("Content") and their compilation are licensed to you subject to the terms and conditions of this Agreement by MindView, Inc., having a place of business at 5343 Valle Vista, La Mesa, CA 91941. Your rights to use other programs and materials included on the CD are also governed by separate agreements distributed with those programs and materials on the CD (the "Other Agreements"). In the event of any inconsistency between this Agreement and the Other Agreements, this Agreement shall govern. By using this CD, you agree to be bound by the terms and conditions of this Agreement. MindView, Inc. owns title to the Content and to all intellectual property rights therein, except insofar as it contains materials that are proprietary to third-party suppliers. All rights in the Content except those expressly granted to you in this Agreement are reserved to MindView, Inc. and such suppliers as their respective interests may appear.

1. LIMITED LICENSE
MindView, Inc. grants you a limited, nonexclusive, nontransferable license to use the Content on a single dedicated computer (excluding network servers). This Agreement and your rights hereunder shall automatically terminate if you fail to comply with any provisions of this Agreement or any of the Other

Agreements. Upon such termination, you agree to destroy the CD and all copies of the CD, whether lawful or not, that are in your possession or under your control.

2. ADDITIONAL RESTRICTIONS

a. You shall not (and shall not permit other persons or entities to) directly or indirectly, by electronic or other means, reproduce (except for archival purposes as permitted by law), publish, distribute, rent, lease, sell, sublicense, assign, or otherwise transfer the Content or any part thereof.

b. You shall not (and shall not permit other persons or entities to) use the Content or any part thereof for any commercial purpose or merge, modify, create derivative works of, or translate the Content.

c. You shall not (and shall not permit other persons or entities to) obscure MindView's or its suppliers copyright, trademark, or other proprietary notices or legends from any portion of the Content or any related materials.

3. PERMISSIONS

a. Except as noted in the Contents of the CD, you must treat this software just like a book. However, you may copy it onto a computer to be used and you may make archival copies of the software for the sole purpose of backing up the software and protecting your investment from loss. By saying, "just like a book," MindView, Inc. means, for example, that this software may be used by any number of people and may be freely moved from one computer location to another, so long as there is no possibility of its being used at one location or on one computer while it is being used at another. Just as a book cannot be read by two different people in two different places at the same time, neither can the software be used by two different people in two different places at the same time.

b. You may show or demonstrate the un-modified Content in a live presentation, live seminar, or live performance as long as you attribute all material of the Content to MindView, Inc.

c. Other permissions and grants of rights for use of the CD must be obtained directly from MindView, Inc. at http://www.MindView.net. (Bulk copies of the CD may also be purchased at this site.)

DISCLAIMER OF WARRANTY

The Content and CD are provided "AS IS" without warranty of any kind, either express or implied, including, without limitation, any warranty of

merchantability and fitness for a particular purpose. The entire risk as to the results and performance of the CD and Content is assumed by you. MindView, Inc. and its suppliers assume no responsibility for defects in the CD, the accuracy of the Content, or omissions in the CD or the Content. MindView, Inc. and its suppliers do not warrant, guarantee, or make any representations regarding the use, or the results of the use, of the product in terms of correctness, accuracy, reliability, currentness, or otherwise, or that the Content will meet your needs, or that operation of the CD will be uninterrupted or error-free, or that any defects in the CD or Content will be corrected. MindView, Inc. and its suppliers shall not be liable for any loss, damages, or costs arising from the use of the CD or the interpretation of the Content. Some states do not allow exclusion or limitation of implied warranties or limitation of liability for incidental or consequential damages, so all of the above limitations or exclusions may not apply to you.

In no event shall MindView, Inc. or its suppliers' total liability to you for all damages, losses, and causes of action (whether in contract, tort, or otherwise) exceed the amount paid by you for the CD.
MindView, Inc., and Prentice-Hall, Inc. specifically disclaim the implied warrantees of merchantability and fitness for a particular purpose. No oral or written information or advice given by MindView, Inc., Prentice-Hall, Inc., their dealers, distributors, agents or employees shall create a warrantee. You may have other rights, which vary from state to state.
Neither MindView, Inc., Bruce Eckel, Chuck Allison, Prentice-Hall, nor anyone else who has been involved in the creation, production or delivery of the product shall be liable for any direct, indirect, consequential, or incidental damages (including damages for loss of business profits, business interruption, loss of business information, and the like) arising out of the use of or inability to use the product even if MindView, Inc., has been advised of the possibility of such damages. Because some states do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation may not apply to you.
This CD is provided as a supplement to the book "Thinking in Java 2nd edition." The sole responsibility of Prentice-Hall will be to provide a replacement CD in the event that the one that came with the book is defective. This replacement warrantee shall be in effect for a period of sixty days from the purchase date. MindView, Inc. does not bear any additional responsibility for the CD.
NO TECHNICAL SUPPORT IS PROVIDED WITH THIS CD ROM
The following are trademarks of their respective companies in the U.S. and may be protected as trademarks in other countries: Sun and the Sun Logo,

Sun Microsystems, Java, all Java-based names and logos and the Java Coffee Cup are trademarks of Sun Microsystems; Internet Explorer, the Windows Media Player, DOS, Windows 95, and Windows NT are trademarks of Microsoft.

Thinking in C: Foundations for Java & C++

Multimedia Seminar-on-CD ROM

©2000 MindView, Inc. All rights reserved.

WARNING: BEFORE OPENING THE DISC PACKAGE, CAREFULLY READ THE TERMS AND CONDITIONS OF THE LICENSE AGREEMENT & WARANTEE LIMITATION ON THE PREVIOUS PAGES.

The CD ROM packaged with this book is a multimedia seminar consisting of synchronized slides and audio lectures. The goal of this seminar is to introduce you to the aspects of C that are necessary for you to move on to C++ or Java, leaving out the unpleasant parts that C programmers must deal with on a day-to-day basis but that the C++ and Java languages steer you away from. The CD also contains this book in HTML form along with the source code for the book.

This CD ROM will work with Windows (with a sound system). However, you must:

1. Install the most recent version of Microsoft's Internet Explorer. Because of the features provided on the CD, it will NOT work with Netscape Navigator. **The Internet Explorer software for Windows 9X/NT is included on the CD.**
2. Install Microsoft's *Windows Media Player*. **The Media Player software for Windows 9X/NT is included on the CD.** You can also go to **http://www.microsoft.com/windows/mediaplayer** and follow the instructions or links there to download and install the Media Player for your particular platform.
3. At this point you should be able to play the lectures on the CD. Using the Internet Explorer Web browser, open the file **Install.html** that you'll find on the CD. This will introduce you to the CD and provide further instructions about the use of the CD.