Thinking in Java

第二版

Bruce Eckel
President, MindView, Inc.

侯捷 / 王建興
合譯

ISBN 0-13-027363-5
90000
9 780130 273635

讀者意見：

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路上找到。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路免費下載了！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路的這種氣度，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

PH PTR
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

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

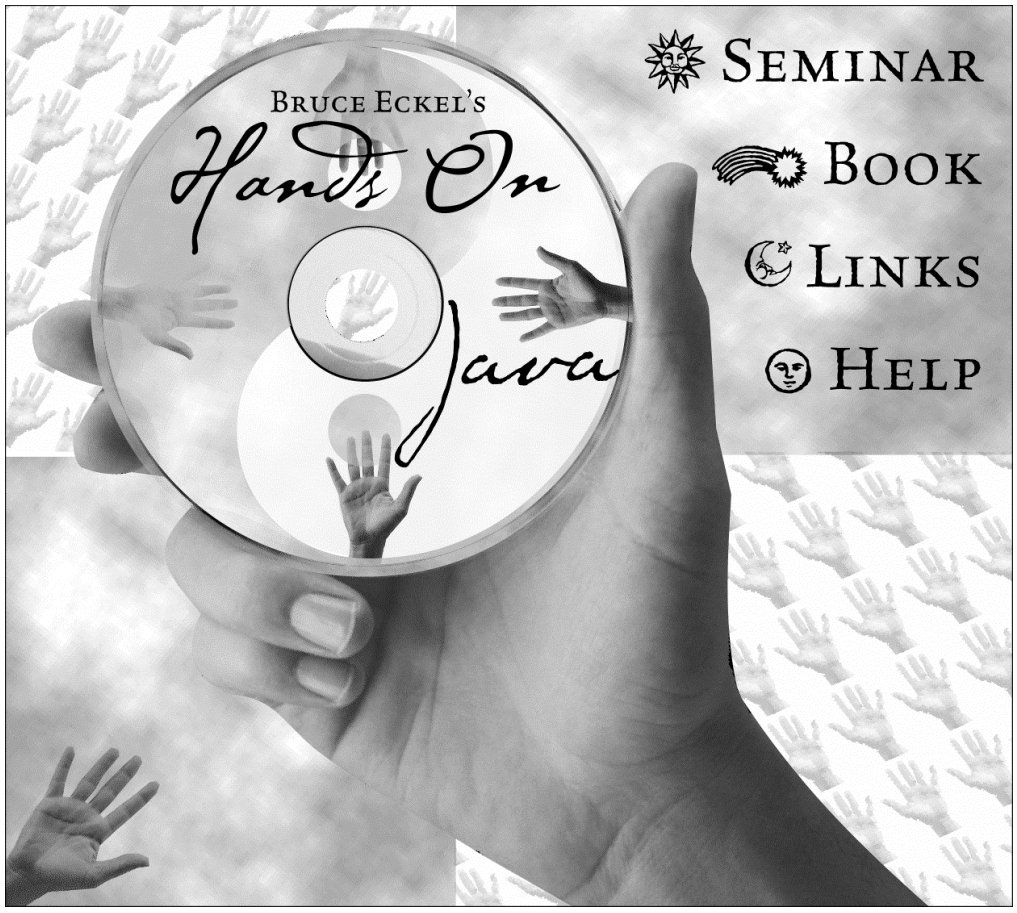
Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

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*

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*

Dedication

To the person who, even now,
is creating the next great computer language

綜覽（Overview）

目錄

我想起我的兄弟 Todd，正在硬體領域與程式設計領域做一次大躍進。基因工程則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。我認爲，這種革命的規模之大，將使電腦革命相形見絀。

但是，然後，我便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革命的到來）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，但這些東西最終都跟陷阱沒有什麼一樣。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？」他們這樣詢問。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，這是我對於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。這是個讓人讚嘆不已的過程，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，而這些毛病原本應該是被我遺漏不察的。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，我從未察覺這些錯誤。我很喜歡這種群體參與過程，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路上實在不錯，可是我希望擁有出版商印出來的、裝訂成冊的紙本。」我曾經努力希望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，只有在吸引到足夠關注時，才考慮製作成紙本形式。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。但是隨著時間的流逝，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設定在「克服程式發展與維護過程中的種種複雜度」上[1] 。當然，許多程式語言在設計時，的確也將複雜度考慮進去，但是總會有被視爲更本質的問題混雜進來。毫無疑問的，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（讓熟悉 C 的程式員能夠

[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。

輕易跨越門檻），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的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，何樂不爲？）。Visual Basic（VB）是另一個例子，它被 BASIC 這個「其實不以擴充性爲設計目標」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「有如能寫不能讀的程式碼（write-only code）」這樣的指責（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，都在複雜度的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「爲程式員降低複雜度」做爲一個堅定的目標。就好像是說：「我們的唯一考量，就是要降低產生穩固程式碼的時間和困難度。」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的很快（雖然目前有許多保證，承諾 Java 總有一天能夠執行的多快多快）。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的時間。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、網路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 library，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性的問題：跨平台程式、動態程式碼改變、甚至涵括安全性議題，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 所帶來的保證卻是很棒的：可以讓我們成爲具有高生產力的程式設計者。

我所見到的種種巨幅改變，有一些是發生在 Web 身上。網路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 語言的設計者仍舊繼續努力使它變得更簡單）。網路程式設計所討論的，便是讓我們以更有效率、成本更低的方式，和其他人通訊，超越舊式的電話媒介（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。

透過所有的努力 － 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）、跨平台程式的執行、跨 Internet 通訊程式開發的大量簡化 － Java 擴展了「人際之間」的通訊頻寬。我認爲，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，因爲我們能夠更輕鬆地和其他人溝通：可以一對一的形式、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，後繼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，但至少存在著可能性。這使我覺得，教育這個語言是一件有意義的事。

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，我不是在將自己和莫札特相提並論。）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，因爲原先內容仍然擺在網站 *www.BruceEckel.com*

上，而且本書所附光碟片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，而且這對作者而言，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（可自網站下載）。基於這些原因，本書理應變得苗條一些才是。

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）。「涵蓋所有的 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，*JSP*s）便是這些議題中涵蓋最廣的一個。伺服端 Java 無疑爲 WWW 的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。此外， Enterprise Java Beans（EJBs）的問世，也使人們希望更輕易地發展資料庫連結、交易處理、安全性考量。這些問題在本書第一版本中，通通被放在「網路程式設計」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：「分散式計算」。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 library。再次提醒你，如果你想要取得舊有的 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所附光碟的內容；新的材料還會陸續增加）。

除了新增的少數 Java 2 語言特性，本書也徹頭徹尾地翻修了一遍。最主要的改變是第九章的群集器（collection），我已將重點改放在 Java 2 的群集器上，並修繕本章內容，更深入鑽進某些更重要的群集器議題之中，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、以及我認爲此一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），並將新的更動加入，然後盡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依然持續有所變化 — 即使不再像以前那樣大刀闊斧。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。如果你希望能夠更輕便地閱讀，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 來閱讀本書內容，並打開背光（譯註：Palm PDA 上一種用於夜間燈光不足時的顯示模示）以免打擾他的太太。我希望這麼做可以幫助他早點進入夢鄉）。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，在火車上閱讀。

Java 2

本書撰寫的當時，Sun 即將發行 *Java Development Kit*（JDK）1.3，並預告 JDK 1.4 中會有的更動。雖然這些版本的主號碼還停留在 1，但是「Java 2」卻是 JDK 1.2 及其後版本的標準名稱。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，加進了許多新特性和美妙的設計。

本書是爲 Java 2 而寫。能夠擺脫過去的沉舊內容，重新以更新的、更精進的語言來撰寫，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片中，如果你要使用 Java 2 以前的版本，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，也不會造成讀者爲了升級而付出金錢。

不過，還是有點小麻煩，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「分散式計算」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，爲了爲數僅數百 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之處。

這張光碟當然包含了本書所有原始碼，此外也包含了本書完整內容的多種電子形式。我最喜歡 HTML 格式，不但因爲速度快，也因爲索引十分完備 — 只要在內容索引表上按下某個欄位，馬上就可以跳到想閱讀的地方。

整張光碟滿滿超過 300 Mega 的內容，是名爲《*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，是想做爲一個獨立產品。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（「控制程式流程」）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得多。我希望這張光碟的附上，能夠使愈來愈多的人被領入 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。如果使用得恰到好處，那麼作爲一種表達的媒介，當欲解決的問題益形龐大複雜，解法益形簡易、彈性。

你不能僅僅將 Java 看成某些特性的集合 － 這些特性之中的某些被支解而被獨立看待時，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 式的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與洞澈；如果你在某個問題上陷入思路的泥沼，便可以將其回饋至腦中模型，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，以及像是「while」之類的迴圈概念…等等。你可以從許多地方學習到這些事情，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「自在地以編程的基本概念來編寫程式」，那麼你便可以使用本書。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。但是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。

有許多參考資料，就像介紹 C/C++語言的功能特色那樣，並不企圖進行更深層的詮釋，僅僅只想幫助所有程式員從其他語言的角度來觀看 Java。這將使得 Java 的主體性向下沉淪。我會試著簡化這些參考資料，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，我開始教授這個語言。程式語言的教學已經變成了我的職業。從 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者，同樣困惑於許多議題。我發現，多年來在「軟體發展會議（Software Development Conference）」上主持 C++（後來變成 Java）專題的工作中，我和其他講者一樣，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，也由於我的教材的呈現方式，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報告。最後，我結束了透過實驗與迭代（iteration，一種在 Java 程式設計中也會用到的技巧）」的學習過程。我根據授課經驗所學來的事物（它讓我可以長時間地快樂教課），發展出一套課程。這套課程採用分離、極易融會貫通的數個步驟來處理學習上的問題，並採取了動手實踐的研討形式（這是最理想的學習方

式）。在其中，我爲每一小部份的課程內容設計了多個練習題。現在我將此一課程置於公開的 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。（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。在我認爲我的這份課程材料足以成爲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與調整，助益甚多。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的整理，藉以引領你平順地前進至下一個主題。最重要的是，此書是爲那些孤單對抗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在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，融於程式語言的教學中。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，切割出合理時數中可以教完、而又容易吸收的材料，並附帶適合於教室中完成的習題。

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。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問題的範圍。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爲教學帶來益處的方式。

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，縮短授課時間及各練習時段間的空檔。這麼做不僅是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，本書才會提供各種 classes 的額外描述。

章節概述

設計本書時，有一個念頭長在我心：人們學習 Java 語言的路線。研討班的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「太過急進、一次含入太多主題」的地方，我也透過了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，避免生出許多困擾。

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，你可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的各個授課時程與練習時段。

第 1 章： **物件導論** *（Introduction to Objects）*
本章提供物件導向程式設計（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（constrtuctors）、物件的生命、物件被建立後置於何處、神奇的垃圾收集器（garbage collector，當物件不再被使用時能夠加以清除）。其他相關議題也有討論，包括異常處理（exception handling）、多執行緒（multithreading）、網路、Internet 等等。你將會看到，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與設計概念。

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授給你，包括：object reference（物件參

照）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），物件被垃圾收集器回收的方式、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，便是 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；其中當然也包括了 Java 的標註式 break 和標註式 continue（這是爲了 Java 不再提供 goto 而設計的），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）。接著便是清除過程的討論，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，垃圾收集器最終會執行任務，釋放該物件所配置的記憶體。我會探討垃圾收集器及其本質。最後我們近距離觀察自動的成員初始化動作、自定的成員初始化動作、初始化順序、static（靜態）初始化、以及陣列初始化。這些細微的物件初始化動作的探討，爲本章劃上美好的句點。

第 5 章： **隱藏實作細目** *（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。最後檢視了 **public**、**private**、**protected** 這幾個關鍵字，並引介「友善的（friendly）」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）觀念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，這是第七章中的主題）。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。你也可以利用所謂的「合成（composition）」，將 objects 嵌入新的 class 之中。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。倚靠自我修練，你或許得花九個月的時間才能夠體會其奧秘所在。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的型別資訊，於是程式得以更易於延伸，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 提供了「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 之中。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 可以使程式碼更優雅、更澄淨。

第 9 章： **物件的持有** *（Holding your Objects）*

在已知的生命週期中維護固定數量的物件，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的物件，而這些時間點只有程式執行之際才能夠確定。此外，除非處於執行時期，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 container library（容器程式庫），讓你藉以妥善儲存你所需要的物件。我的討論包括最簡單的陣列（arrays），以及 **ArrayList**、**HashMap** 之類的複雜容器（或說資料結構）。

第 10 章： **透過異常來處理錯誤** *（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，但有時候有些問題 － 不論是程式員引起，或程式正常執行下自然發生 － 只能夠在執行時期被偵測、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時所引發的種種問題。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，並知道建構式（constructors）中觸發異常時會發生什麼事，了解異常處理程序的放置方式。

第 11 章： **Java 的 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）、輸出（output）。這意謂 I/O（輸入和輸出）在這個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。介於「傳統 I/O」與「全新 Java 式 I/O」之間的差異，也會提及。此外本章也會檢視「將物件串流化（streaming）」（以使物件可被置於磁碟，或可於網路上傳遞）的過程，並討論如何將其重構（reconstructing）。這些都是透過 Java 的「物件串流（*object serialization*）」機制達到。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： **執行時期型別辨識** *（Run-Time Type Identification）*

當你僅有某物件的基礎型別參照（a reference to the base class）時，Java 的執行時期型別辨識（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參照，而能進一步知道該物件的確切型別，將帶來很大用處。通常，此一資訊讓你可以更有效率的執行某些特定動作。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。此外本章也介紹了 Java *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，以及 WWW applets 的開發。同時也會介紹極重要的「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了多顆處理器，否則這種工作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具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： **分散式計算**（*Distributed Computing*）

當你想要開始撰寫運作於網路之上的程式時，一時間，好像所有的 Java 特性及類別庫都一起湧現合作無間。本章探討網路及 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）、遠端函式叫用（*Remote Method Invocation*，RMI）等技術。最後還介紹了 *JINI*、*JavaSpaces*、企業型 JavaBeans（*Enterprise JavaBeans*，EJB）等最新技術。

附錄 A: **物件的傳入與傳出**（*Passing & Returning Objects*）

Java 允許你和物件溝通的唯一方式是，透過 reference 來達成。所以，將物件傳入函式內以及將物件自函式傳回，便存在著某些有趣的結果。這份附錄爲你說明，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、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 *native methods* 即可。Native methods 係以另一種語言（目前僅支援 C/C++）寫成。這份附錄爲你提供了足夠的入門引導，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

我已經從研討班所獲得的經驗中察覺，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，每章末尾都安排了一些習題。

大多數習題都夠簡單，使你得以在有指導者從旁協助的情況下，以合理的時間完成。這可以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，但是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其電子檔解答，收錄於《*The Thinking in Java Annotated Solution Guide*》，僅須小額費用即可自 *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一些介紹。其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與本書內容有關。這張光碟片是個獨立產品，其中含有一週時程之「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，並整合了上百張投影片。由於我的研討班課程係以本書爲基礎，所以這是極爲理想的補充教材。

這張光碟還含有五天時程的精修班課程內容（其主題和個人著重方向有著密切關係）。我們相信，它爲品質樹立了新的標準。

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

原始碼（Source code）

本書所有原始碼，皆宣告爲自由軟體（freeware），以單一包裝的形式進行傳佈。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爲最新版本，上述網站是此一產品的官方站台。你或許可以從其他網站上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。你有權力在課程上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，在大部份媒體上使用本書範例，一般而言就不會帶給你任何麻煩。

在每一個原始碼檔案中，你都可以看到如下述的版權宣告：

```
//:! :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，包括了：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。

我在本書範例程式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*）中找到。此風格似乎也被大多數 Java 開發環境所支援。如果你已讀過我的其他著作，你可能會注意到，Sun 的撰碼風格和我的一致 — 這讓我滿心歡喜，雖然我沒有爲此做了任何事。至於程式碼格式（formatting style），這個主題適合花好幾個鐘頭來爭辯，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，都是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（//!）給標記了起來。因此它們可以輕易被發現，並以自動化的方式被檢測。如果你發現程式碼有錯並回報給作者，我將把你的大名列於將被傳佈出去的原始程式碼中，然後再列名於再版書冊中（並出現於 *www.BruceEckel.com* 網站）。

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 等編號，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 Linux 上取得的 JDK）。

如果你曾學習過 Java 語言的較早版本，而其內容未被本書所涵蓋，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。

還有一件事請你注意，當我需要提及較早的版本時，我不會提微小修正版號。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，依據的材料即爲本書。課堂所授內容，是書中每一個章節挑選出來的材料。每堂課之後皆有指導與練習時段，讓每位學員都能夠得到個別的照料。簡介性的課程，其錄音教材及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，就可以獲得些許的研討經驗。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可以順利走過開發週期 — 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，請使用此一表單來提報訊息，並請附上你的修正建議。如果必要的話，也請附上原本

的原始碼，並標註任何你所建議的修正。對於你的幫助，我謹致以無上的感激。

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 20 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、制服程式臭蟲。這也是 Java 極具威力之眾多特性中的一個。

致謝（Acknowledgements）

首先我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像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，因爲有她的無比耐性、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 總是在正確的時機以及正確的地點出現，清除所有障礙，並做了所有正確的事，成就了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用的 Web 伺服器，都是由他熱心提供。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會議（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（找出了成堆的大臭蟲）、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 的理解也幫助了我認識 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，重新啓發了我對於研討會的可行性想法。

本書的設計、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與設計家（*www.Will-Harris.com*）。當電腦與桌上出版尚未發明之前，他就已經在國中時期玩過所謂的 rub-on letters，並且抱怨我的代數問題含糊不清。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、以及眾多義大利朋友們（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（以及 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 library 撰寫程式範例，及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，程式語言的發端也始於對機器的模仿。

不過，電腦並非那麼冷冰冰的機器，因爲，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。

這一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，先在程式設計與 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* 中取得。

本章所提供的是背景性、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上的學習。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與品質。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、與 C），則在組合語言之上再行抽象化。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來思考問題的命運，因而無法以待解問題的結構來做爲思考的基準。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）」內，例如電腦）與實際待解問題模型（位於問題實際存在之「題域（problem space）」內）之間的關聯性。這裡頭所需要的便是對應（mapping）的功夫，然而這並非該種程式語言的自性本質，同時也使得程式難以撰寫，維護代價高昂。這種副作用，使整個「程式方法（programming methods）」的產業體系於焉誕生。

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，像是 LISP 與 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串列」、「所有的問題都是演算式的（algorithmic）」。PROLOG 則將所有問題都轉換爲一連串的決策。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問題類型，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。

物件導向方法（Object Oriented approach）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類型的問題。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，你還可能動用許多物件，它們無法被類比至題域中的元素。）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；當你閱讀解決辦法的程式碼時，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，而不再是以解法執行之所（電腦）的形式來描述問題。不過，當然了，最終還是會連接回到電腦本身。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，以這種角度來說看似乎不錯 － 它們皆有特性（characteristics）與行爲（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，因而倡言所謂的「多模式（multiparadigm）」程式語言，試圖融合多種不同的解決方法[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 的特性代表著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數，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來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。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特性就是：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。由於型別爲「圓形」的物件，同樣也是型別爲「幾何形狀」的物件，所以「圓形」肯定能接受所有可以發送給「幾何形狀」的訊息。這意謂你可以撰寫和「幾何形狀」溝通的程式碼，並自動處理所有與幾何形狀性質相關的事物。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**，將新的型別導入程式之中，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、有著共同特性與行爲之所有物件」的一部份。

Simula，一如其名，誕生目的是爲了發展模擬程式。例如，在古典的「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。屬於同一類別的眾多物件，除了程式執行時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，和內建型別幾乎沒有什麼兩樣。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，物件便會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示爲獨一無二的個體。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與行爲。

所以，雖然我們在物件導向程式設計的過程中，實際上所做的便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）與共同行爲（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與行爲一樣。箇中差異在於，程式設計者藉著定義 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查（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。

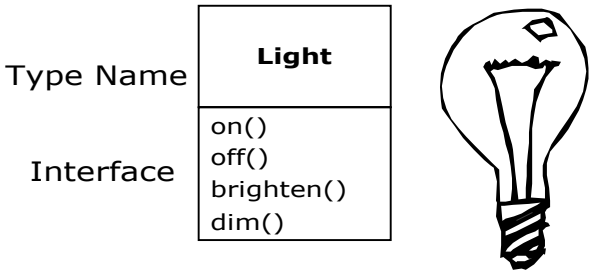
一旦 class 建立之後，隸屬該 class 的物件，你想要多少就能產生多少。你可以操作這些物件，就好像它們是存在於待解問題中的元素一般。確實，

---

[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。

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，建立起一對一的對應。

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，使該物件能夠做些諸如完成一筆交易、在畫面上進行繪製、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。這些程式碼與隱藏的資料，構成了實作（implementation）。從程序式設計（procedural programming）的觀點來看，並沒有太複雜。每個 type 都有一個函式對應於任何可能的請求。當你對某一物件發出請求，該函式便被呼叫。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，執行起對應的程式碼。

本例之中，type/class 的名稱是 **Light**，特定的 **Light** 物件則名爲 **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。欲發送訊息給該物件，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 — 使用預先定義好的 class — 的觀點出發，這種「以物件來進行設計」的型式很漂亮。

上圖是以所謂的 UML（Unified Modeling Language）格式呈現：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）都置於方格的中央，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

將程式開發者依各自的專業領域加以區分，對我們的概念釐清大有幫助。程式開發者可分爲：開發新資料型別的所謂類別開發者（class creators），以及在應用程式中使用他人所開發之類別的所謂客端程式員（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。類別開發者的目標則是打造 classes，並且只曝露出客端程式員應該知道的事物，隱藏其他所有事物。爲什麼？因爲如果加以隱藏，客端程式員便無法使用，這意謂類別開發者可以改變隱藏的部份，不必擔心對其他人造成衝擊。隱藏的部份通常代表著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給毀壞掉。因此，將實作部份隱藏起來可以減少程式臭蟲。不過，實作隱藏（implementation hiding）的觀念也不應該被過度強調。

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。

---

[3] 關於這個詞彙的使用，我得感謝我的朋友 Scott Meyers。

如果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該 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世界之中。

因此，存取權限控管機制的第一個理由便是，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，而非讓使用者用來解決特定問題。這其實是一種對使用者所提供的服務，因爲使用者可以輕易看出，哪些事物對他們來說是重要的，哪些則是可以忽略的。

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，而不需擔心影響客端程式。舉例來說，你可能會想要簡化開發動作，改以較簡單的方式來實作某一特定類別。但是稍後卻發現，你得重新寫過才能改善其執行速度。如果介面與實作二者能夠切割清楚，這個工作便是輕而易舉。

Java 使用三個關鍵字來設定 class 內的存取界限：**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** 之後的所有定義內容，每個人都可加以存取。接續在 **private** 之後的所有定義內容，除了型別開發者可以在該型別的 member functions 中加以存取，沒有其他任何人可以存取。**private** 就像是你與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，這個 class 的繼承者有能力存取 **protected** members，但無法存取 **private** member。稍後會有對繼承（inheritance）的簡短介紹。

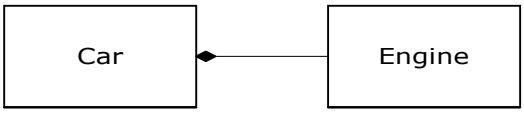
Java 還有一種所謂的「預設（default）」存取權限。當你沒有使用上述任何一個指定詞時，用的便是這種存取權限。有時候這被稱爲 friendly 存取權限，因爲同一個 package 中的其他 classes，有能力存取這種所謂的 friendly members，但在 package 之外，這些 friendly members 形同 **private** members。

重複運用實作碼

Reusing the implementation

一旦 class 開發完成、並經過測試，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與深刻的見解。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 objects。此外你也可以把某個 class object 置於另一個 class 內。我們稱這種形式爲「產生一個 member object」。新的 classes 可由任意數目、任意型別的它種 objects 組成，這些 objects 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」，所以這種觀念被稱爲「合成（*composition*）」或「聚合（*aggregation*）」。合成通常被視爲「has-a（擁有）」的關係，就好像我們說「車子擁有引擎」。



（以上的 UML 圖以實心菱形指向車子，代表合成（composition）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]。）

透過合成關係，程式員可以取得極大彈性。class 的 member objects 通常宣告爲 **private**，藉以讓客端程式員無法直接取用它們。這也使你得以在

[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還是合成。

不干擾現有客戶程式碼的情形下，更動這些 members。你也可以在執行時期改變 member objects，藉以動態改變程式行爲 - 稍後即將探討的「繼承（inheritance）」關係中，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。

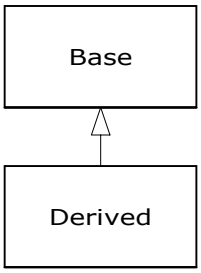
由於繼承在物件導向程式設計中是如此重要，使得它常常被高度地、甚至過度地強調。程式設計新手於是會有一種刻板印象，那就是「應該處處使用繼承」。這會造成誤用，並且導致過於複雜的設計。事實上，在建立新的 class 時，你應該先考慮合成（composition）方式，因爲它夠簡單、又具彈性。如此一來你的設計會更加清晰。有了一些經驗之後，便更能看透繼承的必要運用時機。

繼承：重複運用介面

Inheritance: reusing the interface

object 這一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，而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 有著相似的功能，你還是得被迫重頭建立新的 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。



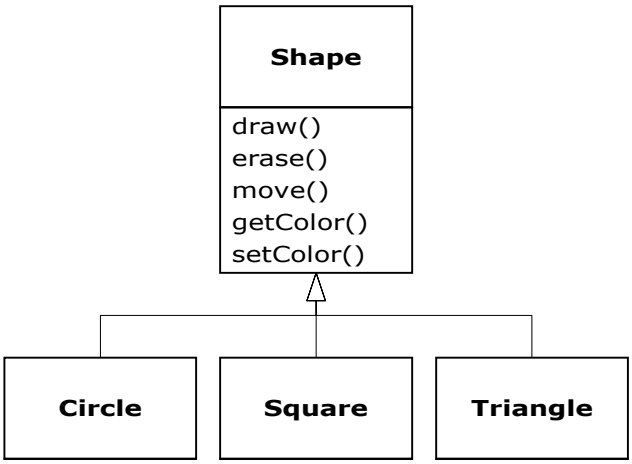
（以上 UML 圖中的箭號，是從 derived class 指向 base class。稍後你便能夠理解，可以存在一個以上的 derived classes。）

type 不僅僅只是用來描述一組 objects 的制約條件，同時也具備了與其他 types 之間的關係。兩個 types 可以有共通的特性與行爲，但其中某個 type 也許包含較多的特性，另一個 type 也許可以處理較多的訊息（或是以不同的方式來處理訊息）。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 共享的特性與行爲。你可以使用 base type 代表系統中某些 objects 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、狀態有關）。透過繼承的使用，你可以建立一個型別階層架構（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與行爲，例如某些形狀可以被翻轉。某些行爲也許並不相同，例如

面積計算方式便不盡相同。型別階層架構（type hierarchy）同時展現了各種形狀之間的相似性與相異性。

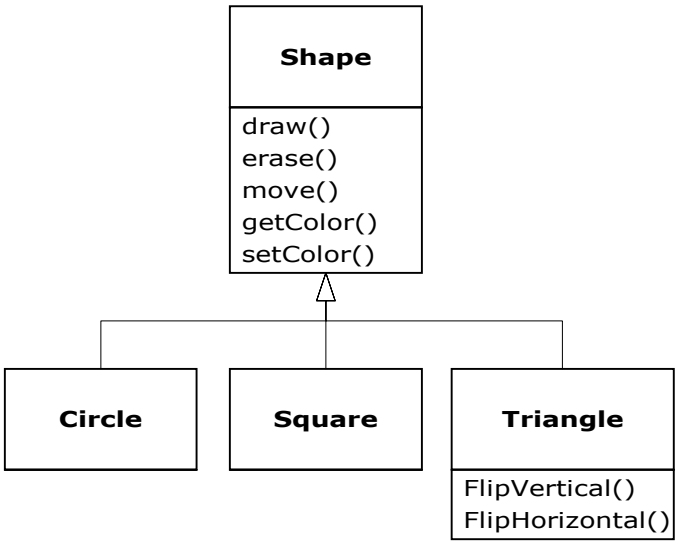


如果我們能以問題原本所用的術語來轉換解答，將會大有益處，因爲你不需要在問題的描述與解答的描述之間，建立起眾多中介模型。透過物件的使用，type hierarchy 便成了主要模型，讓你得以直接自真實世界出發，以程式碼來描述整個系統。是的，對使用物件導向程式設計的人們來說，眾多難以跨越的難關之一便是，從開始到結束太過於簡單了。對於一顆久經訓練、善於找尋複雜解答的頭腦來說，往往會在接觸的一開始被這種單純特性給難倒。

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 members（但 **priavte** members 會被隱藏起來，並且無法存取），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 objects 的訊息，也都同樣可以發送給 derived class objects。由於我們可以透過「可發送之訊息型態」來得知 object 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。

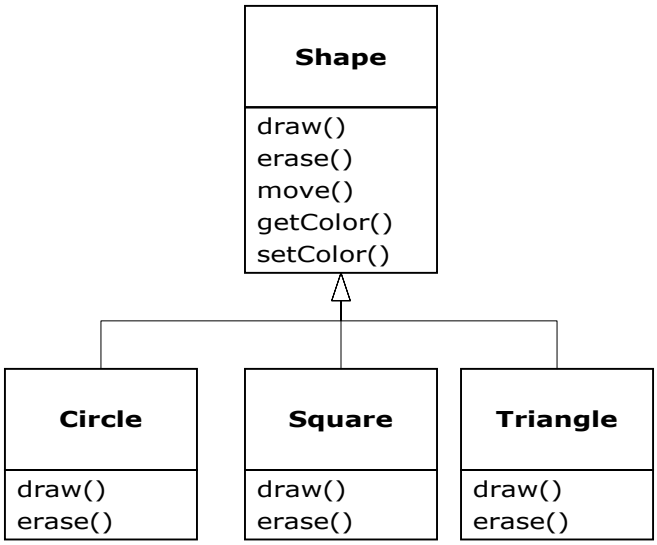
base class 和 derived class 有著相同的介面，而一定有某些實作碼伴隨著此一介面。也就是說，當 object 接收到特定訊息時，還是得有程式碼來執行動作。倘若你只是很簡單地繼承了 class，然後再沒有做任何事情，那麼 base class 的介面所伴隨的 methods，便會原封不動地被繼承到 derived class 去。這表示 derived class objects 不僅擁有與 base class objects 相同的 type，也擁有相同的行爲，這沒什麼趣味。

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。形成差異化的第二種方法（也許

是更重要的方法）便是改變既有之 base class 的函式行爲，這種行爲我們通常稱爲「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些不一樣的事情。」

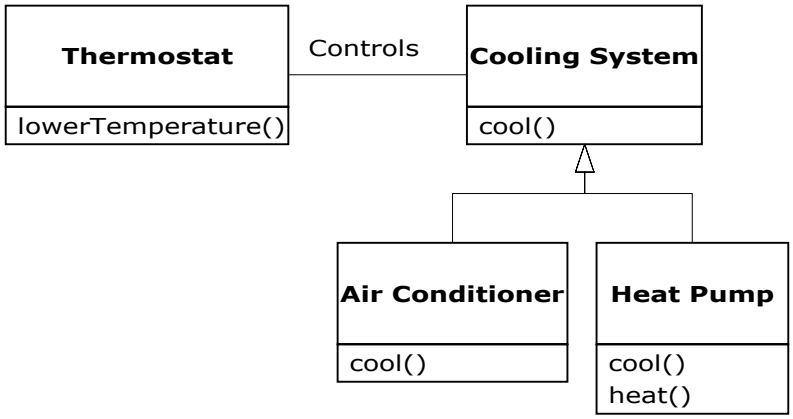
是－佸 vs. 像是－佸

is-a vs. is-like-a relationships

繼承過程中可以進行的動作，仍舊有某些爭論。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，因爲它們的介面一模一樣。這樣得到的結果是，你可以以 derived class object 完全替換 base class object。這可視爲一種「純粹替代（*pure substitution*）」，通常稱爲「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，稱爲「*is-a*（是一種）」關係，因爲你可以說「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，因爲 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。以冷氣機爲例，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，它具備了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。但因爲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，應該改爲「溫度控制系統」，使它得以涵蓋加熱功能 — 然後我們便可套用所謂的「替代法則」了。上圖是個範例，說明在設計領域與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 介面中」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

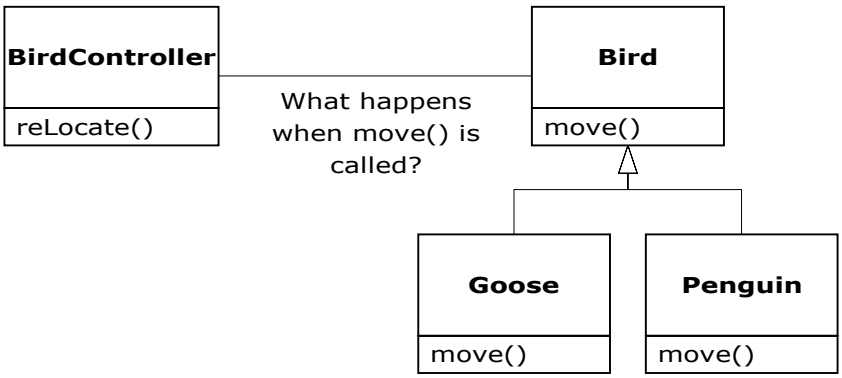
Interchangeable objects with polymorphism

處理 type hierarchy 中的物件時，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，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，以便處理新的狀況，加入新式 types 是最常使用的手法。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 — 只要這些函式僅僅處理泛化的（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 object 時（例如以幾何形狀來看待圓形、以交通工具來對待腳踏車、把鸚鵡看做是鳥等等），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，編譯器在編譯時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。那麼編譯器又會做些什麼事呢？以下圖爲例，**BirdController** object 僅處理泛化的 **Bird** objects，因此它對那些 **Bird** objects 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** objects 究竟是什麼型別，也不需要判斷這些 **Bird** object 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呢？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竅門所在：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，編譯器將叫用動作產生出特定的函式名稱，而聯結器（linker）再將這個叫用動作決議（resolves）爲「欲執行之程式碼的絕對位址」。但是在 OOP 中，程式未到執行時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，必須採用其他的解決方案。

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。當你發送訊息至 object，「應被喚起的程式碼」會一直到執行時期才決定下來。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（無法保證此事

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。

爲了做到後期繫結，Java 使用一小段特殊程式碼來代替絕對形式的呼叫動件。這一小段程式碼會透過儲存於 object 內的資訊來計算函式實體位址（此一過程將於第七章詳細討論）。因此每個 object 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 object，該 object 便知道如何反應。

在某些程式語言中（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）。因此，你不需要加上任何關鍵字，就可以獲得多型（polymorphism）的威力。

回頭想想幾何形狀的例子。整個 classes 族系（擁有一致化介面的所有 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，並在程式碼中略去型別細節，僅與 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，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*）。

倘若以 Java 來撰寫 method（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，所以獨立於任何它所繪製或擦拭的特定 object type 之外。如果我們在程式的其他地方用到了 **doStuff()** 函式：

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

```
doStuff(t);
doStuff(l);
```

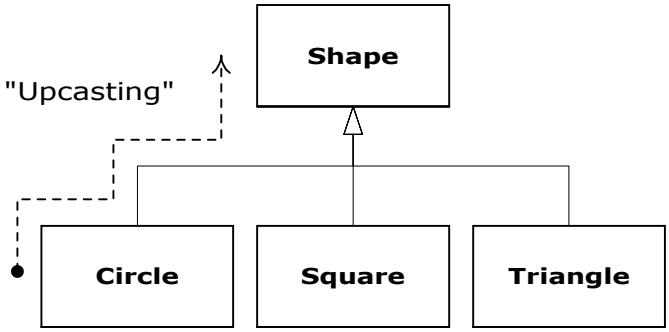
那麼當呼叫 **doStuff()** 時，不論 object 實際型別爲何，都能夠運作無誤。

這是個令人感到驚奇的手法。再看看下面這行程式碼：

```
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 視爲其所屬之 base class」的過程稱爲「向上轉型（*upcasting*）」。*cast* 這個字靈感來自於鑄造模型時的塑造動作，*up* 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（cast）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「對實際型別的執著」救贖出來的關鍵。請看 **doStuff()** 內的函式碼：

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

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

object 的實際型別，而那會帶來極大麻煩，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() 的內容實在讓人感到神奇，不知怎麼地，事情竟然會自己搞定。呼叫 **Circle** 的 **draw()**，執行起來的程式碼，和呼叫 **Square** 或 **Line** 的 **draw()**，執行起來的程式碼完全不同。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 object 時，即便動用了向上轉型（upcasting），object 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** 來標示該種 class 爲「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。這是強迫某種特定設計的好工具。

你也可以使用關鍵字 **abstract** 來描述目前尚未被實作完成的 method，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 method 僅能存在於抽象 class 之中。當 class 被繼承，抽象 method 必須被實作出來，否則新的 class 仍然是個

抽象 class。建立抽象 method，讓你可以將這個 method 置於介面之中，毋需被迫爲該 method 提供或許毫無意義的程式內容。

關鍵字 **interface** 又更進一步發揮抽象的觀念，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「介面與實作分離」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等重要，本節剩餘篇幅涵蓋了這些議題。

一個極爲重要的議題便是 object 的生成（created）與毀滅（destroyed）。Object 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 objects 置於 stack（這樣的 objects 又稱爲 automatic 變數或 scoped 變數）或靜態儲存區中，於是程式撰寫時便決定了 object 的儲存空間與壽命。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時明確知道 objects 的數量、壽命、型別。如果你企圖解決的是較一般化的問題（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 objects。在這種方式之下，除非等到執行時期，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 objects，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。動態法有一個前提假設：適用於 object 較爲複雜的情況，因此不論儲存空間的找尋或釋放時所需的額外負擔，都不致於對 object 的生成造成重大衝擊。動態法帶來的彈性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 object，都得使用關鍵字 **new** 來動態產生實體。

還有一個議題，就是 object 的壽命。在那些「允許 object 生成於 stack 內」的程式語言中，編譯器會判斷 object 應該存活多久，並可自動加以消滅。但如果在 heap 之中生成 objects，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消滅 objects。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「垃圾收集器（garbage collectioncollector）」機制，當 object 不再被使用，會被自動察覺，並被消滅。垃圾收集器十分便利，因爲它可以減少你需要考量的因素，以及你必須撰寫的程式碼數量。更重要的是，垃圾收集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）。

本節的其餘篇幅中，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

如果你不知道將動用多少 objects，或者不知道這些 objects 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 objects。如何才能夠得知該爲這些 objects 準備多少空間呢？這其實永遠得不到答案，因爲這些資訊只有在執行時期才能獲得。

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 object。這個新型別的 object 儲存著一些 reference，代表其他 objects。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。這種新的 object 通常被稱爲容器（*container*），有時也叫群集（*collection*）。由於 Java library 以不同的意義使用後一個術語，所以本書採用「容器」一詞。這個新的 object 會適當擴展自己的容量，以便容納你置入的所有東西。你不需要知道將置入多少 objects 於容器之中，是的，只要產生容器，它會自行料理細節事務。

幸運的是，好的 OOP 語言往往都會伴隨一組容器，做爲 package 中的一部份。在 C++ 中，容器是 Standard C++ Library 的一部份，有時候也稱爲 Standard Template Library，STL。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 standard library 中提供了許多容器。在某些 library 之中，通用型容器能夠完善符合所有需求，其他 library（例如 Java）則提供了諸般不同的容器，符合各色需求。例如 vector（在 Java 中稱爲 **ArrayList**）爲所有元素提供了一致性的存取方式，linked list（串列）則提供任一位置上的插入動作。你應該從中挑選符合需求的型別。容器類別還可能包括：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與取出。通常它們都會提供元素安插函式，以及函式取回函式。不過，元素的取出動作較爲複雜，因爲「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 object，其工作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」與「對容器進行存取動作的程式碼」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 與 1.1 時，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。如果單一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**。這兩種序列可以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，卻有截然不同的差異。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，如果你將元件安插至序列中央位置，對 **LinkedList** 來說花費的代價就明顯少於 **ArrayList** 了。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（對大多數的 objects 而言，這是基本觀念）。倘若你所處的程式開發環境，有著其他因素所引起的額外負擔，那麼，介於 **ArrayList** 與 **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。

單一根源的繼承體系
The singly rooted hierarchy

OOP 之中有個議題，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，單一根源的繼承體系可以帶來許多好處。

單一根源繼承體系中的所有 objects 都有共通介面，所以最終它們都屬於相同的 type。在另一種設計中（C++ 所提供），你無法確保所有 objects 都隸屬同一個基礎型別。從回溯相容的觀點來看，這麼做比較符合 C 的模型，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，而種便利性在其他語言身上卻是內建的。你的 library 內可能有一些不相容介面，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上出發，Java 這種方式通常會帶給你更大的生產力。

單一根源繼承體系（一如 Java 所提供的）可以保證所有 objects 都擁有某些功能。在整個系統裡，你因此知道可以在每個 object 身上執行某些基本操作。單一根源繼承體系，再加上「在 heap 之中產生所有 objects」，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）。

單一根源繼承體系也使垃圾收集器（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，然後垃圾收集器便可發送適當訊息給系統中的每個 object。如果缺乏「單一根源繼承體系」及「完全透過 reference 來操作物件」的系統特性，垃圾收集器的實作就會十分困難。

由於所有 objects 都保證會有執行時期型別資訊（run-time type information，RTTI），所以你不會因爲無法判斷 object 的確切型別而陷入動彈不得的僵局。對於異常處理（exception handling）之類的系統層級操作行爲而言，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，及對容即性支援

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 library 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 library，而且幾乎可以滿足所有需求。

向下轉型 vs.模版/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。單一根源繼承體系意謂「萬事萬物皆爲 **Object**」，因此，如果容器可以裝納 **Object**s，也就可以儲存任何事物。這使得容器很容易被重複運用。

欲使用這樣的容器，只需簡單地將 object reference 加入即可，稍後可以將它們取回。由於容器僅裝納 **Object**s，所以當你將 object reference 加入容器時，會向上轉型爲 **Object**，因而遺失自己的身份。當你將它取回，你所得到的僅僅是個 **Object** reference，而非你所置入的 object 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 object 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** 是否爲 **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，因爲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 objects 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查動作，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 object 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爲我們量身訂製的 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++ 極爲重要的組成，部份原因是 C++ 缺乏單一根源的繼承體系。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，因爲若是使用了單一根源繼承體系，又再加上這種功能，可能會變得…呃…有點笨拙。不過目前有一份相關提案，其中採用和 C++ template 極爲相似的語法。

清掃的難題：誰該負責打掃？

The housekeeping dilemma: who should clean up?

每個 object 爲了生存，都需要動用資源，尤其是記憶體。當 object 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，清除 objects 似乎不是什麼困難的問題：首先產生 object，然後使用之，最後它就應該被摧毀。這不難，但你可能遭遇更爲複雜的情況。

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、錄影帶出租系統等的核心一樣適用）。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 object，即是完成了清除動作。

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。它可能記錄著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 object，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）操作第二容器中的 objects。

現在問題更加困難了：何時才是消滅這些 objects 的適當時機？當你用完某個 object 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 objects」的系統中，這件事情顯得格外複雜。

Java 的垃圾收集器被設計用來處理「釋放記憶體時可能會有的種種問題」（其中並未包含物件清除動作的其他面向問題）。垃圾收集器會「知道」物件何時不再被使用，然後便會自動釋放該物件所用的記憶體。這種方式（合併了「單一根源繼承體系」以及「完全從 heap 中配置記憶體」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。只需做極少的決策，只有極少的障礙需要克服。

垃圾收集器 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，這個代價就是執行時期的額外負擔。一如先前所述，在 C++ 中，你可以選擇在 stack 上生成 objects，它們會被自動清除（但不具備彈性，無法在執行時期依需要而動態產生）。在 stack 上生

成 objects，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 objects，代價可就高昂多了。即使完全繼承同一個 base class，完全以多型方式來處理函式呼叫，同樣需要付出少量代價。垃圾收集器之所以特別形成癥結，問題在於你永遠不知道垃圾收集動作何時開始、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，例如在極爲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厲。

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語言功能，也不希望在那些「程式員可能會選擇 C 的場合中」因爲 C++ 所引進的新功能而影響到 C++ 的使用。這個目標達到了，但也使得以 C++ 進行設計時需要付出「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，對於解決極大多數程式問題而言，Java 是較好的選擇。

異常處理：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 library 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 object，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，異常處理機制會採用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地忽略，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」提供了一種方法，是的，不再是只能選擇退出（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 object 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。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）」，這種觀念便被稱爲「多執行緒（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，真正做到平行（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，便是讓程式設計者毋需考量實際機器上究竟存在幾個處理器。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，但當資源共享時卻有著隱約的問題。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。所以整個過程是，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 object 層次上提供，因此一個執行緒便以一個 object 來表示。Java 也提供有限的資源鎖定功能，可以鎖定任何 object 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 object。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

object 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 object 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，object 便又能夠復活，而且仍然具備上次執行行時的狀態。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）」，讓你可以很簡單地將 object 儲存於磁碟，並於稍後取回。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複雜的支援。

Java 與 Internet

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中革命性的一大步。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（網路衝浪）、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，其經濟價值與結果究竟爲何。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。

主從式電算技術
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，散佈給某些人或是某些機器。主從概念的關鍵核心便在於，資訊貯存處的位置集中於一點。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，而資訊與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*）。位於遠端機器之上、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，卻必須同時服務多位用戶。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）。如果用戶端軟體需要更動，他們必須重新建立、除錯、然後安裝在用戶端機器上。整個過程可能超乎你想像的複雜與費力。如果想要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，都可能帶來重大的影響。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。然而主從式計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路中。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問題就好了。

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，不單從伺服器上擷取檔案，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。

有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。此即所謂的「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）、核取鈕（check boxes）、選取鈕（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，以及將資料提交（submit）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。提交的內容會告訴 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，該目錄通常被命名爲「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，因爲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到所有目的。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與網路的擁擠程度有關。而且 CGI 程式的初始化動作先天上就比較慢。Web 的初始設計者沒有預見到，網路頻寬（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，因爲得先產生一個個 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 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 頁面用以通知你這個錯誤，並將此頁面回傳給你；於是你得回到前一頁面，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，可以執行十分龐大的工作。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與解法。

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爲瀏覽器加入新功能。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，也不是爲特定網站而做的事。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 頁面中的部份文字，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，所以這個問題不算太嚴重。

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，更明確地說是用於建立更豐富、更具互動性的圖形化使用介面（GUIs）上。script 語言的確可能解決掉用戶端程式開發過程中遭遇的百分之八十的問題。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 之類更複雜的解決方案之前，或許你應該先考慮使用 script 語言。

最常被討論的瀏覽器 script 語言包括：JavaScript（和 Java 毫無關連；其名稱只是爲了搭上 Java 的浪潮而已）、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）都提供內建支援。此外，比起其他的瀏覽器語言，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。不過如果你對 Visual Basic 或 Tcl/Tk 已經駕輕就熟，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，內建了安全性功能、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與其 libraries，得以優雅地處理傳統語言中諸多未能處理好的問題，像是多執行緒、資料庫存取、網路程式開發、分散式計算。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，這是它優美的地方之一 — 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，所以在送出請求至伺服器之前與之後，你都可以盡情地在用戶端工作。例如你不必透過網路送出請求表單，然後才發現所填的日期或參數錯誤。用戶端的電腦可以快速繪出資料，不必等待伺服器完成圖形的繪製後，再將圖形傳回。你所獲得的不僅僅是高速度和高回應能力，也可以降低網路流量和伺服器負載，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（所謂的 JAR），使得所有必要模組都能夠以壓縮形式被包裝起來，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，也比較減少對伺服器的存取）。另一個重要因素是 Web 站台的回應速度。還有一個因素，那便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。當你點擊了某個 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受限。Java 的設計理念中，也將 applet 的執行侷限於受到安全保護的 sandbox（沙盒）內。在

sandbox 裡頭，applet 無法對磁碟寫入資料，也無法存取 sandbox 外部的記憶體內容。

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。

所謂的「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。這個方法乃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「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，而安全性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性限制，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，低成本的網路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 爲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 sandbox 的牢籠。

數位簽證遺漏了一個重要問題，也就是人們在 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，需要多久時間才能夠發現這個破壞？可能數天，也有可能長達數週。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

主從式架構的問題，目前是最常被使用的解決方案就是 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古典問題，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 瀏覽器與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路時，我們便將此網路稱爲 intranet（企業用的內部網路）。Intranet 相較於 Internet，能夠提供更高的安全性，因爲你可以實際控制對公司內的伺服器的存取。若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與 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中自動形成的角落。假若你沒有把握你的程式在 Internet 上會執行於什麼平台，那麼你就得格外小心，以免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，你得負責你自個兒程式碼的品質，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中的程式碼，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挑選，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

截至目前的討論，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的「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，如果用戶端更聰明些，伺服端可以透過 Java 或 scrip 語言只傳回原始資料，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，尤其因爲它們可以消除不同能力之瀏覽器而衍生的問題。（譯註：術語補充：servlet 一詞由 server 和 let 兩個字合成。Java 世界常以 let 字尾表示小東西，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：伺服端

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，讓你在更短的時間內開發穩固的程式，速度之快勝過任何一種程式語言。

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 專案方法時，許多人都可能感到困擾。一旦你知道每項事物都被想成是 object，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*），是一組程序與竅門，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，便已有許多系統化的 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、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，最重要的事莫過於此。別放棄，做起來其實很簡單。大多數分析、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，成功地分析與設計[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。請牢記，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

[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。

時、或甚至程式執行時，才能夠發覺。正因如此，快速完成分析與設計動作，並實作出系統上的測試，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與實作階段之前，必須周延地考慮任何瑣碎細節，並加了解。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所在。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與實作階段之前，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可用來解決此類問題。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）。企圖在短期間內獲得初步實作，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「對於後繼的分析、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及文件，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 objects 是什麼？（如何將案子切割爲眾多組成？）
2. objects 的介面是什麼？（你得發送什麼樣的訊息給每個 object？）

確定了 objects 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內容和文件，但以上兩點是最基本的需求。

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

階段 0：計劃

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（事實上所有的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（我想有朝一日我會寫本小說），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，當我書寫與電腦有關的事情時，結構對我來說十分清楚，不需特意思考。儘管腦袋只在半意識狀態，我仍舊可以妥善地將作品結構化。好的，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。就像好萊塢電腦中所謂的「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、然後啓程。」

階段 1：我們在做什麼？

Phase 1: What are we making?

前一世代的程式設計（即所謂程序式設計，procedural design）將此一階段稱爲「產生需求分析與系統規格」。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：「爲了滿足需求，程式做了哪些事情（但並不討論如何做到）」。需求分析就是你和客戶之間的契約（即使該客戶也服務於同一公司，或甚至是某些 objects 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」這兩個問題的探究。由於這二者都需要取得人際共識（而且通常隨著時間改變），所以我認爲它們愈少愈好 — 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，不得不將其擴展爲更大的文件，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，這可以燃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 會找出系統的關鍵性特色，這些特色可以揭示某些即將動用的根本類別（fundamental classes）。這一類本質性的描述式解答，回答的問題如下[9]：

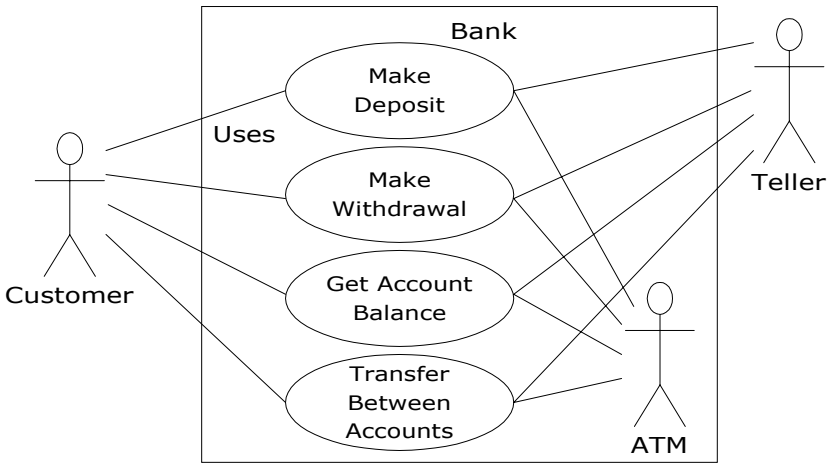
- 誰將使用本系統？
- 系統參與者會透過此系統做些什麼事情？
- 參與者如何透過此系統做到該件事情？
- 如果其他某人正在做這件事，這件事情會以什麼其他方式運作？或問「同一參與者是否具有另一個不同目標？」（顯示出變異情形）
- 透過此系統來進行該件事情時，可能會有哪些問題？（顯示出異常情況）

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爲。這些情境都被稱爲「腳本（scenario）」，use cases 可被視爲腳本的集合體。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？」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？」

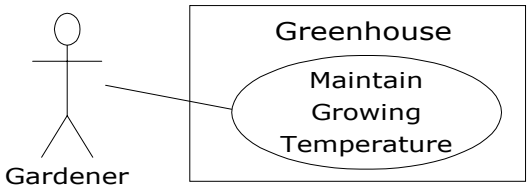
use case diagrams（使用案例圖）刻意設計得十分單純，避免你過早陷於系統實作細節之中：



上圖的每一個棒狀小人兒，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。介於參與者與 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」，產生需求規格。你會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性影響」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：以些許幾段文字來描述系統，同時找尋其中的名詞與動詞。名詞可以關聯至參與者、use case 中的脈絡（例如大廳）、或是在此 use case 中被操作的人工製品。動詞則可關聯到介於參與者與 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，名詞與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「名詞與動詞」這種技巧僅能做爲腦力激盪工具，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與運用，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 member functions 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 objects 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 objects，提供服務給此一 class 的 object。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）？前者會想知道用哪化學材料來製造鞭炮，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，不僅是希望讓你的 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過仔細，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。在群體中不依賴電腦地即時完成它，最是有效。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 objects 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 objects。爲他們提供設計方案初稿的這一次經驗，是我在使用 CRC card 之前，最爲成功的一次諮詢經驗。我們討論著 objects 應當用什麼方式彼此溝通，然後擦掉某些 objects，以其他 objects 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] 來爲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，然後好好沉思一番時。UML 之外的另一種替代方案是，「object 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爲，重要到必須以專屬圖示來加以表示時（例如在控制系統中），這些符號就十分有幫助。當資料成了具支配性的影響因素時（例如資料庫），你可能也需要描述系統與子系統內的資料結構。

當你完成了 objects 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 objects 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 objects 就好了。雖說若能在整個過程中愈早發掘出來最好，但 OOP 提

---

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

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

供了足夠的結構，因此即使晚一點才發掘也不會造成什麼負面影響。事實上，在程式發展的過程中，objects 的設計應該發生於五個階段。

Object 設計的五大階段

object 的設計並不侷限於程式撰寫時期。事實上 object 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 object 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）。object 當然也有它們自己的樣式，在了解、使用、重複運用 object 的過程中，便會一一顯現出來。

1. object **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，包括外在的因子與邊界、系統中的元素重複情況、最小概念單元，便可能發掘出 object。如果你已有一組 class libraries，那麼某些 objects 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. object **的組合**：實地發展 objects 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 object 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 object 的更多需求。就像學習過程一樣，你的 object 會逐步演化。由於得和系統中的其他 object 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. object 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 object 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 objects 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。

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。Objects 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

這個階段會進行初次轉換，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與系統規格（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）」的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個星期（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。即使在分析與設計之後，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。

當你完成了所設定的全部功能，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「同意」的行業。）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；採行開放原碼（open-source）策略的專案，在這種反覆往返、高度倚賴回饋訊息的環境中，正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。影響重大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，使它承擔某些品質上的欺瞞行爲。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澈的了解，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」之資源的關鍵。

「事事正確無誤」所指的，並非單只是讓程式根據需求和 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，程式結構在面臨生命週期中無法逃避的更動時，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此類「持續性修改動作」的支援，是的，由 objects 所形成的邊界，會主動保持結構免於毀壞。物件導向程式語言讓你可以進行更動（這對程序性程式來說可能是很嚴重的事），而你的程式碼不致於處處崩裂。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，結合了程序式方法中「結構不足」的缺點，往往導致混亂的系統，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，更勿需害怕）。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。

取得平衡

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，來爲自己領路。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。而這正是財富之所在啊。

Extreme programming (XP)

從唸研究所的時候開始，我便斷斷續續研讀過不少分析與設計技巧。*Extreme Programming*（XP）是我見過最激進、最讓人感到愉快的一種觀念。在 Kent Beck 所著的《*Extreme Programming Explained*》（Addison-Wesley, 2000）書中，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 強烈主張完整的程序（process），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的生產力與穩定度。

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」。這種想法使得測試在潛意識裡頭被認爲是件低優先權的工作。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 讓測試工作和原碼撰寫平起平坐（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 應該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。前者像是一紙清單，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」的過程中漏失了。

「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觀看程式語言的發展，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。「這個資料型別是否被妥善運用」和「這個函式是否被正確呼叫」都是由編譯器與執行系統所進行的測試。我們已經看到了這些測試內建於程式語言中的結果：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，內建測試功能的安全網絡會告訴你，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）」一樣。XP 卻背棄這種想法，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「在一塊空間裡頭擺一堆工作站」，不應該在其中樹立室內設計者最喜歡的隔板。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]。（那麼當然就得有個管理者，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）擴音裝置在內。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（但是管理者無法想像，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 爲什麼成功

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多的協助。Java 的目的是實用；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（亦即解域空間中的術語），而是以題域空間中的術語（像是「把索環放進箱中」）來描述你的解法。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 libraries 發揮最大模程效能

開發程式的最快途徑，就是使用現成的碼：library（程式庫）。Java 的主要目標之一，便是讓 library 的使用更容易。「將 library 轉換爲新的 data types（classes）便是 Java 達成此一目標的手段。因此，引入 library，意謂著將新的 types 加到程式語言裡頭。由於 Java 編譯器會留意 library 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 library 的用途，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存在。即使存在，你還是會加以忽略。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 － 也就是擦去了小程式與大程式之間的複雜度界線。如果你只是想寫一個「Hello World」的小程式，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，不論是小程式或大程式，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會是：「我要如何才能夠讓我的上司、同事、部門、同儕們都開始使用 objects 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。首先是訓練與範例教學，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，可以參考以下所建議的實踐準則：

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與撰碼課程。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 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 Standard Java libraries 更是如此）。當你能夠把既有的 libraries 當做基礎，建立 objects 並加以使用，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 libraries 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 重寫既有的程式碼

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成 objects 形式，你可以使用 Java 原生介面（Java Native Interface）和 C 或 C++ 程式碼接軌。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。遷移到 Java 平台上所需付出的成本，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（或是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 平台上所需付出的代價，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 libraries 並加以購買，而不是試著自己重新開發這些 libraries，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」以及「可能導入的新設計環境」所引起的生產力損耗。訓練與顧問諮詢無疑地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會湧現的掙扎。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，即便大家還只是在學習 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，每天所寫的程式碼或許更少）。

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 所提供的額外安全性功能，傳統上是犧牲諸如 C++ 語言所具備的效能而來的。諸如「hot spot」之類的技術及編譯器上的技術，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 libraries，它們通常都已經由供應廠商最佳化了，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，先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方式可以施行於底層，使上層程式碼不需要隨之更動。只有在這些方法都束手無策時，你才需要更改設計。由於效能在設計中佔了極重要的地位，所以效能必須成爲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 *native methods* 與之接軌，這個主題涵蓋於本書附錄 B。

常見的設計錯誤

一旦你的開發團隊採用 OOP 與 Java，程式員通常會經歷一連串常見的設計錯誤。通常是因爲在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。這可能是因爲沒有在公司內培養出專家，也可能是因爲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++，而且是如此成熟，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 libraries、component models（組件模型）、code-generation tools（程式碼產生工具）結合在一起，藉以解決視窗環境下（例如 Microsoft Windows）與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 的 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路、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、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++ libraries（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。有許多流傳說法（我從一些原本使用 C++ 後來轉投 Java 陣營的開發團隊聽來的），認爲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 libraries，將那些 Java 程式碼轉換爲 C++。）

使 Java 足堪適用於大多數專案的關鍵，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可以逼近 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的附錄中，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 爲何如此與眾不同、Java 又爲何如此格外出眾、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，它們用於表達的詞彙，太偏電腦，不像是你要解決的問題所用的術語。

在你所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。啊，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」的所謂 objects（而非任何電腦表述方式），以及那些在 objects 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「重複運用既有的 libraries」的方式解決掉了。

---

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

2: 對象 對物皆物件

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 語言。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，萬事萬物都是物件（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 object，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 object 呢？

這一切在 Java 中都大大地簡化了。所有事物都被視爲 object，單一固定的語法便可通用各處。不過，雖然抽象上你可以把所有東西都「視爲」object，但用以操控 object 的識別字，實際上卻只是其「reference（參照）」而已[1]。Object 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 object。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 object，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++的 reference 與 Java 的 reference 相較之下，某些地方還是有著頗爲重要的差異。我那時剛退出 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，這些人應該會是 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」。作者認爲，Java 的物件識別字實際上就是一個 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是在撒謊，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爲 object 進行初始化。

所有 objects 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 object 去。通常我們會透過關鍵字 **new** 來做。**new** 的意思是「請給我一個新的 object」。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「讓我成爲一個新的 **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 objects？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（**Register**s）。這是速度最快的儲存場所，因爲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。作爲一個程式員，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 object reference 儲存於 stack 內，但卻不能將一般的 Java object 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），用來置放所有的 Java objects。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 objects，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，你得爲這樣的彈性付出代價：從 heap 中配置空間，比從 stack 中配置（假設你真的可以在 Java 中像 C++一樣地自 stack 產生 object 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 object 內的特定成員設爲靜態，但 Java object 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。因爲常數是不會改變的，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 的形式中，objects 被轉換爲一連串的 bytes。這些 bytes 通常用來傳送到另一部機器。在 persistent objects 的形式中，objects 被儲存於磁碟，所以即使程式結束了，這些 objects 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 objects 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 objects。Java 提供了所謂「輕量級的永續性（lightweight persistence）」，新的版本有可能提供更完善的解決方案。

特例：基本型別（primitive types）

在 object type 之外，還有一種 types 應該被特別對待。你可以把這類 types 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：這便混入了非物件導向語言的性質）。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空間的大小，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。

Primitive type	Size	Minimum	Maximum	Wrapper type
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「外包類別（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，外包類別便可派上用場。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包類別，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操作，是很相像的。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「method 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 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，或是在記憶體被初始化之前便加以使用（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 的主要目標之一便是安全。許多不斷困擾 C/C++程式員的種種問題，Java 便不再重蹈覆轍。Java 保證陣列一定會被初始化，而且程式員對陣列的存取，無法逾越其範圍。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 objects 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，陣列中的每一個 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 object 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。

你不需要摧毀 object

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 object，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 */
  }
}
```

編譯器會認爲變數 **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。因爲，Java 設計者認爲，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 objects 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 object，即便離開了生存空間，該 object 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** object，仍然會繼續佔用記憶體。如果單看上面這段程式碼，無法存取到這個 object。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 object」的那些個 references 是如何地被四處傳遞與複製。

經由 **new** 所產生的 object，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 object 的時候，object 的確能夠供其使用。更重要的是，在 C++裡頭，你得在用完 objects 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 objects 繼續存在，究竟是什麼機制使這些 objects 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「垃圾收集器（*garbage collector*）」機制，它會逐一檢視所有透過 **new** 所產生的 objects，並在這些 objects 不再被參考到時（譯註：也就是不再有任何 reference 指向它們時），知道它們的確不再被參考到了。然後，垃圾收集器便釋放這些 objects 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 objects 的產生就好了。所有 objects，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 objects，那麼究竟什麼東西用來決定某一類 objects 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 object 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** 來產生屬於此一型別的物件：

```
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 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）。

欄位（fields）和方法（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 class objects、將訊息發送給 objects），你可以將兩種成員放進去：data members（資料成員，有時稱爲欄位，*field*），以及 member functions（成員函式），後者通常稱爲 *methods*（方法）。data members 可以是任何型別物件，只要你可以透過它的 reference 來和它溝通就行。data members 也可以是基本型別（也就是不具備 reference 者）。如果 data members 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 data members 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 object 都持有用來儲存 data members 的空間；不同的 objects 彼此之間並不共享 data members。下面這個 class 擁有一些 data members：

```
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 object：

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

你可以指定其 data members 的值，但首先你得知道如何取得（參考到）一個 object 的 member。那就是在 object reference 之後緊接著一個句點，然後再接 member 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 object 所含之其他 object 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;
```

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 member functions（亦即 methods）。如果想了解 member functions 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）

當某個 class member 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：

Primitive type	Default
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 內的一個 member」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 data members，百分之百會有初值（這是 C++不同於 Java 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，無法套用於一般區域變數（也就是並非「某個 class 內的 member」）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（就和在 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）

方法（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*（方法）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。

Java 中的 method，決定了某個 object 究竟能夠接收什麼樣的訊息。你將在本節學到，定義一個 method 其實是很簡單的。method 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 */
}
```

method 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 method 的種種資訊，則化爲 argument list（引數列）中的型別和名稱。對 class 而言，method「名稱加上引數列」的組合，必須獨一無二，不能重複。

Java 的 methods 僅能做爲 class 的一部份。只有透過 object，而且它能夠執行某個 method，你才能呼叫該 method[2]。如果你對著某個 object 呼叫了它並不具備的 method，編譯時候就會得到錯誤訊息。如果想要呼叫某個 object 的某個 method，只要指定 object 名稱，緊接著句點符號，再接續 method 名稱和引數列即可，例如 **objectName.methodName(arg1, arg2, arg3)**。假設有個 method **f()**，並不接收任何引數，回傳型別爲 **int**。如果有個名爲 **a** 的 object，允許你呼叫它所擁有的 **f()** method，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 method」的行爲，稱爲「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爲「將訊息發送給物件」的方式。

---

[2] 稍後你會學習所謂的 **static** method（靜態方法），它們可透過 class（而非 object）被呼叫。

引數列 (The argument list)

外界傳給 method 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 object 的形式出現。所以引數列中必須指定每一個想要傳入的 object 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 object reference[3]。傳入的 object reference 型別必須正確。如果引數爲 **String**，那麼你傳入的就必須是個 **String** object 才行。

假設有個 method，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 method 告訴你，如果要儲存指定的 **String** object，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 method，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()** method，那是 **String** 提供的眾多 methods 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 method」，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 method 時，你可以決定任何你想要回傳的型別。但如果這個 method 並不打算回傳任何東西，你應該將回傳型別指定爲 **void**，例如：

---

[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 object，其實是傳遞 object reference。

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

當回傳型別爲 **void** 時，關鍵字 **return** 就只是用來離開 method。我們不必等到執行至 method 最末端才離開，可以在任意地點回返。但如果 method 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 methods」之 objects 的組合。這些 object methods 可以接受其他 objects 作爲引數，也可以發送訊息給其他 object。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 method 中做判斷，藉以執行更細膩的動作。對本章而言，發送訊息的方式就很足夠了。

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了同樣的名稱，該如何區分二者，使它們不會牴觸呢？C 裡頭的這個問題格外嚴重，因爲程式之中往往充滿許多難以管理的名稱。C++的 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。爲了解決這個問題，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。爲了讓 library 內的名稱不致於和其他名稱相混，Java 採用與 Internet 電腦域名（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 電腦域名反過來寫，以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 libraries，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，但卻未明示 *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 都得有一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 library 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 library。（在其他語言中，library 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 standard Java library 中的種種組件。這個 library 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電腦域名。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 object 的外觀長相及行爲舉措。除非以 **new** 來產生 object，否則並不存在任何實質的 object。產生 object 之際，儲存空間才會配置出來，其 method 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 objects，或不存在任何 object 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 method 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 field 或某個 method 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** method，或是取用其 **static** data。一般情形下，你得產生某個 object，再透過該 object 取用其 data 和 method。所以，non-**static** data/method 必須知道它們隸屬於哪一個 object，才有辦法運作。由於使用 **static** method 前並不需要先產生任何 object，所以在 **static** method 中不能「直接」取用 non-**static** data/method。如果只是單純地直接呼叫 non-**static**

method，而沒有指定某個 object，是行不通的，原因是，non-**static** data/method 都得和特定的 object 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 object 有所關聯，「其存在只是爲了 class」的 data 和 method。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 data member 或 method 的定義前，就可以使它們成爲靜態。以下便可以產生一筆 **static** data member，並將它初始化：

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

現在，即使你產生兩個 **StaticTest** class objects，對 **StaticTest.i** 來說，仍然只有一份。產生出來的那兩個 objects，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 object 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** members 是行不通的（但對於 **static** members 卻比較好，因爲這種寫法可以更強調所參考的對象的個 **static** member）。

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

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

相同的模式可以推廣到 static method。你可以透過 object 來取用某個 static method，和取用其他普通的 method 沒什麼兩樣。你也可以經由以下特殊語法來取用：**ClassName.method( )**。定義 static method 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** 的 **incr()** method 將 **static** data **i** 的值累加一。你可以用一般方式，透過 object 來呼叫 **incr()**：

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

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

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

某個 data member 被宣告爲 **static** 之後，勢必會改變其建立方式（因爲 **static** data member 對每個 class 而言都只有一份，而 non-**static** data member 則是每個 object 各有一份）。對 **static** method 來說，差別反而沒那麼大。**Static** method 的最重要用途之一，就是讓你可以在不建立任何 object 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()** method 的定義，做爲程式的執行起點。

就像任何 methods 一樣，**static** method 可以產生或使用其型別所衍生的具名物件，所以 **static** method 常常被拿來當作「牧羊人」的角色，負責看管眾多隸屬其型別的一整群 objects。

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 standard library 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，都被視爲註解。

---

[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。以上程式碼所需要的觀念，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。

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

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

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 的列表上找到所有伴隨 Java 的 class libraries。請點選 java.lang，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，所以此一 library 內的所有 classes 都不需要 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 library 才能夠使用它。假若你不知道某個 class 位於哪一個 library 內，或者如果你想同時瀏覽所有 classes，你可以在 Java 文件中選擇「Tree」，然後便可以看到伴隨 Java 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** library 了。爲了使用 **Date** class，你可以寫 **import java.util.*** 加以含括（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** object。因爲它是靜態的，所以你不必做任何事情，**out** object 便已存在，你只管使用便是。我們能對 **out** object 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 methods，數量相當多，本書稍後將加以討論。此刻我們只對 **println()** method 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。在那個 class 中，必須含有一個 **main()** method，其標記式（signature）必須是：

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

其中的關鍵字 **public**，表示此一 method 是要公開給外界使用的（第五章有更詳細的說明）。傳入 **main()** 的引數，是個 **String** objects 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()**，因爲 **args** 被拿來儲存「命令行引數（command line arguments）」。

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** object，然後直接傳給 **println()**。當這個敘述句執行完畢，產生出來的 **Date** object 再也不會被使用了，因此垃圾收集器（garbage collector）便會在適當時機將這個 object 所佔據的空間收回。此一 object 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、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一下其上所附的文件，決定如何編譯與執行。

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，請你下載本書的原始程式碼（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，便表示你並沒有妥當安裝好 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。

這整個過程，便是本書每一個程式的編譯與執行過程。不過你還會看到，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* 網站上關於本書的網頁，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 * 做爲每行的起頭，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
```

文件化註解

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。如果文件與程式碼二者分離，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，用以標示這種「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出，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。這個工具使用了 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 method 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。我們也可以期望甚至要求，所有 Java libraries 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 /** 爲首的註解之中，並以 */ 作爲結束。Javadoc 的使用有兩種主要型式：內嵌式 HTML，或是文件標籤（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 的註解必須恰好出現在 method 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，輸出結果中是看不到它們的。（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，client 端也只能看到這些。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排版，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。請千萬不要使用諸如**<h1>**或**<hr>**之類的標題（headings）標籤於內嵌的 HTML 中，因爲 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: 參考其他的 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到其他 class 的說明文件。javadoc 會自動爲**@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
```

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

「類別（class）文件」所用標籤

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*（參見第八章）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 文件中。

@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 文件中。

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。

「變數（variable）文件」所用標籤

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

「方法（method）文件」所用標籤

除了**@see** 標籤和內嵌 HTML 的方式外，針對 method 所做的說明文件，還可以使用描述其傳入參數、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，直到遇到新的標籤，才算結束。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；同樣可以延續數行。

@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 method 在執行過程中發生問題時，便可以將異常擲出，告訴外界發生了這個錯誤。當你呼叫某個 method 時，雖然

最多只可能出現一個異常，但 method 可能產生任何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：所謂完整是指包含 package 名稱）。**description** 則是用來說明，在什麼情形下呼叫這個 method，會產生此一型別的異常。

@deprecated

這個標籤標示，此一 method 已被更新的功能替換，建議使用者不要再使用它，因此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 method，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 對於類別（classes）命名有一個不成文規定：將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，第一個字母都採用大寫。例如：

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

幾乎所有名稱，包括 methods、fields（member variables）、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，你應該必須體認，類別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 libraries 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了撰寫一個簡單的 Java 程式所應具備的相關知識。你也已經能夠對 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，所有的範例都只停留在「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...」，該怎麼進行呢？Java 所提供的此類程式設計基本行爲，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 method 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** 執行這個程式。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與執行。
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()** method，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行傳入的三個引數。爲此，你得對代表命令行引數的 **Strings** 陣列進行索引。

8. 將 **AllTheColorsOfTheRainbow** 這個範例，改寫成正式的程式，編譯後執行。

9. 找出 **HelloDate.java** 的第二個版本，也就是本章中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. 在習題 10 中，以內嵌 HTML 的方式加入一個 HTML 項目列表。

12. 使用習題 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:程式流程的控制
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，物件與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與簡化。

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，能教導你盡早學會在 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 methods，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（+）、減法與負號（-）、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，此即所謂「副作用（*side effect*）」。這些會更改運算元之值的運算子，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 objects 身上（但這種應用頗易令人迷惑）。除此之外，**String** 類別也支援 '+' 與 '+=' 這兩個運算子。

優先序（Precedence）

所謂運算子優先序，定義出多個運算子同時出現於單一運算式（expression）時，該運算式的核定（evaluate）方式。Java 對於核定動作的進行順序，依循特定的規則。「先乘除，後加減」是最容易記住的一條規則。至於其他規則，很容易被遺忘，所以你應該使用小括號，明確指定核定順序。舉例來說：

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

便和以小括號加以區分的同一述句，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*），並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;**）。

基礎型別的賦值動作相當直覺。因爲基礎型別儲存的是實際數值，而非 object reference。當你進行基礎型別的賦值動作時，會將某值複製到另一個值身上。例如，對基礎型別寫下 **A = B**，**B** 的內容便會被複製到 **A**。如果你接著修改 **A** 的值，**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 object 時，你所操作的其實是它的 reference。所以當你「將某個 object 指派給另一個 object」

時，實際上是將其 reference 從某處複製到另一處。這意謂，如果對 objects 寫下 **C = D** 這樣的式子，會造成 **C** 和 **D** 都指向原先 **D** 所指的 object。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（instances，**n1** 和 **n2**）在 **main()** 中被產生。每個 **Number** 實體內的 **i** 都被賦予不同之值，然後，**n2** 被指派給 **n1**，然後改變 **n1** 的內容。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** 的內容，同時也更動了 **n2** 的內容！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 在指派過程中被覆寫了，實際上也就是遺失掉了；垃圾收集器（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。如果在這個例子中你不希望別名現象發生，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，毋須將 **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，直接操作物件中的欄位會導致混亂，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（賦值）動作會帶來令人意想不到的結果。

呼叫 Method 時的別名（aliasing）問題

當你將物件傳入 method，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 method 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 也使用簡略標記方式，讓某種運算動作和指派動作同時進行。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 methods 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 methods：**nextInt()**、**nextLong()**、**nextFloat()**、**nextDouble()**，產生多個不同型別的亂數。

將模數運算子（%）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：

```
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）與遞減（decrement）

Java 和 C 一樣，充滿著許多能夠帶來便捷的方法。這些便捷方法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。

在許多便捷方法中，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。以面是個實例：

```
//: 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），如果以後序形式爲之，會在運算執行之前便先擷取其值。所有運算子中，會引起副作用的，除了那些帶有指派動作的運算子外，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（!=）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 **boolean** 的比較結果。想當然爾，其輸出結果當然先是 **true** 然後是 **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** 然後 **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 object 的內容是否相等，又該如何？你得使用 **equals()** method。任何一個 objects（當然不含那些能夠正常運用 ==和 != 的基礎型別）都會擁有這個 method。以下便是這個 method 的使用方式：

```
//: 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));
  }
} ///:~
```

其結果便如你所預期，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：結果是 **false**。這是因爲 **equals()** 的預設行爲是拿 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爲。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。

大多數 Java library classes 都覆寫了 **equals()**，所以它們都會比較 objects（而非 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，使用了關係運算子與邏輯運算子：

```
//: 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** 值之上。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。你可以看到上例因爲這個原因而導致失敗的部份，已用 `//!` 標示起來成爲註解。緊接於其後的算式則使用邏輯比較式來產生 **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** 基礎型別來替換 **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 method 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完成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。如果可以免除評估算式中的所有部份，並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），產生結果值 1。位元運算子 NOT（~），也稱爲「一補數（*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。

位元運算子與邏輯運算子使用相同的運算符號。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，較高位元便會補上 0；如果原值是負數，較高位元便會補上 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*）：不論原值爲正或負，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**，在位移動作發生之前，其值會先被晉升成 **int**，運算結果也同樣會是 **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 位元長）。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，它們會被先晉升爲 **int**，然後進行右移；但是當它們被指派回去時，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，所以產生正確的行爲。

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();
  }
} ///:~
```

最末的兩個 methods，**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()** 不會換行，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。

三元的 if-else 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 *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（此一三元運算子的濫觴）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的時候有所警惕 — 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，事先經過周詳的考慮。

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 上的 operator+

在 Java 中，有個運算子提供了很特別的用法：之前你已經見過的 + 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），對程式員而言反而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，編譯器會將雙引號所括住的字元序列轉化爲 **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。在 Java 中這句話仍然成立。

在 C 和 C++中，下面是一個極爲常見的錯誤：

```
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爲 **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**，編譯器則預期此處應該是個 **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，因爲 **x = y** 是合法算式，但這卻不是你所想像的情況。

C 和 C++ 中還有一個類似問題：將位元運算子 AND 和 OR 誤爲對應的邏輯運算子。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 ==，很容易只鍵入一個符號而非兩個。Java 編譯器也會杜絕此一問題，它不想讓你的漫不經心破壞了編程大計。

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爲另一個型別。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** 轉換爲 **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（包括所有修飾詞）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，因爲編譯器會在必要時候自動將 **int** 晉升（promote）爲 **long**。不過，爲了表明某種觀點，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。在 Java 中，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 — 彼時你得冒著遺失資訊的風險。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*）時就毋需有明確指示，因爲新型別能夠容納來自舊型別的資訊，絕不會有任何資訊遺失。

Java 允許你將任意基本型別轉型爲其他任意基本型別。然而 **boolean** 是個例外，它完全不接受任何轉型動作。class 型別不允許轉型動作。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，同一型別族系的物件之間可以轉型。是的，**Oak**（橡樹）可以被轉爲 **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，當你將某個常數值置於程式中，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。當這種情形發生，你得提供某些額外資訊，透過「爲常數值搭配某些字元」的形式，引導編譯器做出正確的判斷。以下程式碼展示了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）表示式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、十六進制表示），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）表示法係以 **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**。在科學與工程中，'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}。但是當 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與工程領域，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，這個慣例亦被 C、C++、Java 採用。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來做非必要的補充。下面這種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。但是如果寫成這樣：

```
float f4 = 1e-47f; // 10 to the power
```

編譯器會順理成章地將指數視爲 **double**s。所以，由於少了附加的字元 **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。而且由於指派的目的地是較小的型別，有可能遺失資訊。一般來說，算式內出現的最大資料型別，是決定該算式的運算結果的容量大小

---

[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' 搞混過。'E' 僅僅只是代表指數（exponential），代表所使用的數值系統的基底 － 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。』

的依據之一；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。

有關優先序（precedence）議題

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（胃潰瘍患者正是 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型式的運算。

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**，因而必須明確地將它轉回原先型別。操作 **int** 時，毋需動用轉型，因爲每個運算元都已經是 **int**。但千萬別因此鬆懈，進而認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** 的複合指派動作（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）運算有著相同的結果。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。

程式執行時的流程控制

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，它仍然是最爲權宜的方式）。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**（它們令非零值爲真，零值爲僞），Java 卻不允許這麼做。如果你想要在 **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()** method 示範 **if-else** 的使用。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 method 即將傳回之值（如果該 method 的回傳型別不爲 **void** 的話），並令該值立即被回傳。上例中的 **test()** method 可以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** library 中的 **static** method **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 與 **do-while** 之間的唯一差別在於：**do-while** 中的述句至少會執行一次，即使算式一開始就被評估爲 **false**。在 **while** 中，如果條件句一開始就是 **false**，迴圈中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之值，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** 所定義的地方，正是它被使用的地方，也就是在 **for** 迴圈的控制算式裡，而不在成對大括號所標記的區段起始處。**c** 的可見範圍正落在由 **for** 所控制的算式中。

諸如 C 之類的傳統程序式語言，要求所有變數都必須被定義於區段起始處。這麼一來，編譯器在建立區段時，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。

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（因而會將 **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** 會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，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** 的確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。不過，我們現在所談的 **goto**，卻是原始碼層次上的跳躍，而這正是其所以惡名昭彰的源頭。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 — 在極少數情況下，**goto** 其實是流程控制的最佳方法。

雖然 **goto** 是 Java 的保留字，但是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，於是回到外層迭代。狀況 2，**continue** 會將執行點移至內層迭代的起始處。狀況 3，**continue label1** 會同時中斷內層與外層迭代，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，**break lable1** 也是跳脫一切約束，移動至 **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之中的 **prt()** method。

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，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。

本例輸出結果如下：

```
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可以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 method，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 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** 會回到最內層迴圈的頂端，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 與 **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。請牢記，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，直到遇著 **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 與 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，即爲 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** 以利乘法進行，而乘積亦是一個 **double**。這同時也意謂，爲了完成加法，**'a'** 必須被轉換爲 **double**。最後所獲得的 **double** 再被轉爲 **char**。

將 **double** 轉爲 **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** 轉爲整數值，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「包括」，小括號意指「不包括」。）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，只須在命令行（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 與 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]範圍內者）的值（包括正數與負數）小於 1。所以上述表示式的結果的 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。

已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 methods 放進程式中。
3. 請將標題爲 “if-else” 和 “return” 兩小節中的 test() 和 test2() methods，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。再試著改用 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 for 迴圈（nested for loops）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: 初始化與清理
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 library 時，如果不知道如何初始化 library 元件（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*）」的觀念。建構式是一種特殊的 method，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收集器（garbage collector）：當物件不再被使用，垃圾收集器能自動釋放那些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

你可以這麼想像：爲你所撰寫的每個 class 都發展一份名爲 **initialize()** 的 method。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此一 method。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 具備有建構式，

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此一 method 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 members 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 method。C++ 的解法似乎是最簡單也最符合邏輯的，所以也被 Java 採用：建構式的名稱和 class 名稱相同。對於這種「會在初始化動作進行時被呼叫」的 method 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 object 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 object 之前，它絕對會被適當地初始化。

請注意，「每個 methods 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 method 一樣，你也可以爲建構式提供引數，藉以指定 object 生成方式。上例可以輕易做點修改，使其建構式接受引數：

```
//: 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);
  }
} ///:~
```

建構式引數讓你得以爲 object 的初始化動作提供引數。例如，假設 class **Tree** 擁有一個「接收單一整數引數」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** object：

```
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** object。

建構式能夠消解極多問題，並使程式碼更易於閱讀。以前述程式碼片段爲例，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()** method。在 Java 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 method，因爲它沒有回傳值。這和「回傳值爲 **void**」有極大的差別。回傳 **void** 時，一般 method 並不回傳任何東西，但是一般的 method 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。

Method 的多載化（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 object，便是給予對應的儲存空間一個名稱。Method 則是動作的名稱。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 objects 和 methods。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個別、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 － 每個函式都得有一個獨一無二的名稱。

在 Java（和 C++）中，建構式是「method 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 object，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（所謂「預

設（*default*）」建構式，亦稱爲無引數（*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 object 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 methods 共同存在，多載化（overloading）便成了不可或缺的重要關鍵。Method overloading 對於建構式來說是必然產物，而當它運用於一般 methods 時，一樣能帶來便利。

以下例子同時示範了「運算子」和「一般 methods」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** object 可以有兩種生成形式：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「預設建構式[1]，*default constructor*」），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()** method，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 允許我們賦予這兩個 methods 相同的名稱。

區分多個 methods

如果多個 methods 具有相同的名稱，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 methods 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：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 methods？

即便是引數順序不同，都足以區分兩個 methods（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：無引數建構式（no-arg constructors）。但由於預設建構式（default constructors）這個詞彙已行之多年，所以我延用之。

```
//: 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()** methods 都有相同的引數，但擺設順序不同，這使它們得以有所區別。

基礎型別（primitives）的多載化

基礎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。以下程式說明了當基礎型別被置於多載化的 method 時，究竟會引發什麼現象：

```
//: 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 被視爲 **int**，所以如果某個多載化的 method 接受 **int** 做爲引數，該 method 便會被呼叫起來。至於其他情況，如果你所提供的資料，其型別小於 method 的引數，該資料的型別便會被晉升。**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 method 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 methods 會接受較小的基礎型別值。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 method 引數列來做爲多載化的區分呢？爲什麼不依據 methods 的回傳值來區分呢？」舉個例子，下面兩個 methods 有著相同的名稱、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 method 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 method：

```
f();
```

Java 如何才能判斷應該喚起哪一個 **f()** 呢？程式碼的讀者又要如何才能明白這一點呢？基於此點，你無法以回傳值型別做爲多載化 methods 的區分基準。

預設建構式（Default constructors）

一如先前所述，預設建構式（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預設建構式。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 object 並呼叫預設建構式 － 即使你沒有自行定義任何預設建構式。如果沒有預設建構式，我們就沒有任何 method 可供呼叫，也就無法建構出我們想要的 object。請注意，如果你自行定義了任何一個建構式（不論有無引數），編譯器就不會爲你自動合成預設建構式：

```
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。當你沒有設計任何建構式，編譯器說：『你需要建構式，讓我來爲你產生一個吧』。但是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；如果你沒有撰寫預設建構式，你必然是刻意如此』。

關鍵字 this

假設你有兩個同型的 objects，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 objects 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 object **a** 或 **b** 呼叫的呢？

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 object」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()** 中，此一引數便是「正被操作中的 object reference」。所以上述兩個 methods 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 method 之中，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字（名稱）。爲了這個原因，關鍵字 **this** 因應而生。這個關鍵字僅用於 method 之中，能取得「喚起此一 method」的那個 object reference。你可以採取任何其他 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 method 中呼叫同一 class 的另一個 method，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 object 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 object，所以我們可以輕易在同一個 object 身上執行多次操作。

在建構式中呼叫建構式

當你爲某個 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 object」或「目前的 object」，並自動產生「目前的 object」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：

```
//: 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起始處，否則編譯器會發出抱怨。

這個例子同時也展示了 **this** 的另一用途。由於引數 **s** 和 data member **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 data member **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 method 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** method。它的意思是說，對於這個 method，沒有所謂的 **this**。你無法在 **static** methods 中呼

叫 non-**static** methods[2]（反向可行）。不過倒是可以經由 class 本身來呼叫其 **static** method，無需透過任何 object。事實上這也正是 **static** method 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** method 置於 class 之中，可以提供「對其它 **static** methods 和 **static** fields」的存取。

有些人認爲 **static** methods 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** method 時，並非以「送出訊息給 object」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** methods，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。是的，即使 Smalltalk 也具備所謂 class methods（等同於 **static** methods）這樣的東西呢。

清理（Cleanup）：終結（finalization）與垃圾收集（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 libraries 時，在用完 objects 之後，如果只是將它「棄而不顧」，有時候會帶來危險。當然啦，Java 提供了垃圾收集器來回收那些不再被使用的 objects 的記憶體空間，但是請你想想某些不尋常的情況。假設你的 object 並非經由 **new** 獲得記憶體。垃圾收集器只知道釋放那些經由 **new** 配置得來的記憶體，所以它不知道如何釋放你的這個 object 所佔用的「特殊」記憶體。爲了因應這種情況，Java 提供一個名爲 **finalize()** 的 method，你可以爲自己的 class 定義此一 method。以下說明它被期望的運作方式。當垃圾收集器打算開始釋放你的 object 所佔用

[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。

的儲存空間時，會先呼叫 **finalize()**，並且在下一次垃圾收集動作發生時才回收該 object 所佔用的記憶體。所以，如果你選擇使用 **finalize()**，它便讓你得以在「垃圾收集時間」執行某些重要的清理動作。

這中間有個潛在的編程陷阱。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++ objects 絕對會被摧毀（在無臭蟲的程式中），而 Java objects 並不絕對會被垃圾收集器回收。換一個說法：

> 垃圾收集（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 object」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 method 來執行清理動作。舉個例子，假設 object 在產生過程中將自己繪製於畫面，如果你沒有自行動手將它從畫面上拭去，這個畫面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 object 被垃圾收集器回收之前，其影像會先從畫面上被拭去。如果 object 沒有被回收，影像就持續存在。所以請牢記第二點：

> 你的 objects 可能不會被回收

你可能發現，某個 object 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，而垃圾收集器完全沒有被啓動以釋放你的 objects 所佔據的記憶體，那些空間便會在程式終止時一次歸還給作業系統。這是好事，因爲垃圾收集器會帶來額外負擔，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()** 做爲一般用途的清理工作。那麼，它究竟爲何而存在呢？

第三句金玉良言就是：

垃圾收集動作（garbage collection）只回收記憶體

也就是說，垃圾收集器存在的唯一理由，就是要回收那些在你程式中再也用不著的記憶體空間。所以任何和垃圾收集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有關。

這是否意味如果你的 object 內含其他 objects，在 **finalize()** 中應該明確釋放那些 objects？嗯，答案是：no，垃圾收集器所留意的是「object 記憶體」的釋放，不論 object 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 object 以「產生物件」以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 objects，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 *native methods*（原生方法）時。所謂 native methods，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 native methods 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 和 C++ 函式，所以你得在 **finalize()** 中以 native methods 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 object，使用者必須在「打算進行清理動作」時，呼叫清理用的 method。聽起來很簡單，但這麼做和 C++ 的解構式（destructor）觀

念稍微有點抵觸。在 C++ 中，所有 objects 都會被摧毀，或者說，所有 objects 都「應該」被摧毀。如果某個 C++ object 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 object 產生地點所在的大括號範圍結束前自動發生。如果該 object 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）時被呼叫。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 object 的其它部份也不會被清理。這類程式臭蟲很難追蹤。

相對之下，Java 不允許你產生 local objects，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 object，因爲垃圾收集器會爲你釋放空間。所以，從簡化觀點來看，你可以說因爲有了垃圾收集機制，所以 Java 不需要解構式。但是當你隨著本書的內容前進，你會發現，垃圾收集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 method，這個 method 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，就是用來觀察垃圾收集的過程。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收集動作的種種敘述：

```
//: 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** objects，並且在垃圾收集器啓動的時間點上，停止產生 **Chair**s。由於垃圾收集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** 的旗標來標示垃圾收集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 objects。這兩個旗標都應該在 **finalize()** 中加以設定。垃圾收集動作一旦發生，便會呼叫 **finalize()**。

另有兩個 **static** 變數：**created** 和 **finalized**，用來記錄 **Chair** objects 的數目以及被垃圾收集器終結的數目。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** object 序號 47 被終結，旗標會被設爲 **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，因爲它終究會回收到序號 47。

每次迭代所產生的 **String** object，只是爲了當做額外的配置空間，藉以激發垃圾收集器的啓動。因爲垃圾收集器會在認爲可用記憶體數量不足時才開始執行。

欲執行此一程式，你得經由命令行提供必要的引數：”gc” 或 ”finalize” 或 ”all”。如果指定 ”gc”，會令程式呼叫 **System.gc()**（強迫執行垃圾收集器）。如果指定 ”finalize”，會呼叫 **System.runFinali-zation()**，理論上會讓每個未被終結的 objects 都被終結。如果指定 ”all”，會使得上述兩個 methods 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，說明垃圾收集與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 objects 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 objects 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 objects 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 finalizers。不過，此一設計最後被認爲是有問題的，所以這個 method 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收集與終結的議題。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，「finalizers 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()** 被呼叫，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 objects。

請記住，無論是垃圾收集動作或終結動作，都不保證一定會發生。如果 Java Virtual Machine（JVM）並未面臨記憶體耗盡的情境，它不會浪費時間於垃圾收集上，這很聰明。

死條件（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。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 object 的「死亡條件（*death condition*）[3]」的檢查。

當你對某個 object 不再感興趣 － 也就是它可以被清理時 － 這個 object 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 object 代表某個已開啓的檔案，那麼該檔案在此一 object 被回收前，應該先被關閉。只要這個 object 的任何部份未被適當清理，你的程式中便存在難以尋得的臭蟲。**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 {
  boolean checkedOut = false;
  Book(boolean checkOut)
    checkedOut = checkOut;
```

[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。

```
  }
  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** objects 都被預期在它們被回收之前先被登錄（checked in）。但是在 **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，這會是一個棘手的程式臭蟲。

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，使得垃圾收集器啓動），最終還是有可能找出遺失的 **Book**。

垃圾收集器（garbage collector）的運作方式

如果你以往所用的程式語言，從 heap 空間中配置 objects 的代價十分高昂，那麼你可能直覺地假設，Java 這種從 heap 空間中配置所有事物（基本型別除外）的體制，也得付出高昂成本。然而垃圾收集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 空間中配置 objects 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 object 都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 與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 object，這條履帶便往前移動。這意味 object 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，並且在不久之後耗盡資源。獲得成功的關鍵在於垃圾收集器的介入。收集垃圾的同時，它也會重新安排 heap 內的所有 objects，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。垃圾收集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。

你得了解另一種不同的垃圾收集器（GC）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 object 皆內含所謂的參用計數器，每當某個 reference 連接至某個 object，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，因爲垃圾收集器必須逐一走訪一長串的 objects，並且在發現某個 object 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 objects 彼此之間形成循環性的交互參考，那麼即便整群 objects 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 objects，對垃圾收集器來說是件極度龐大的工程。Reference counting 通常被拿來說明某種類型的垃圾收集動作，但似乎從來沒有被真正用於任何 JVM 身上。

爲求更快的速度，垃圾收集動作並非以 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 object，才算是生命尚存的 object。整串追蹤過程可能會涵括好幾層 objects。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 objects。針對每個你所找到的 reference，你都必須再鑽進它所代表的 object，然後循著該 object 內含的所有 references，再鑽入它們所指的 objects。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 object 都必定是存活的。請記住，這種作法並不存在「自我參考」群組的問題 — 那些 objects 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*）」垃圾收集機制。至於它如何處理它所找到的存活的 objects，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收集機制），然後將所有找到的 objects 從原有的 heap 複製到另一個 heap，並將所有垃圾捨棄不顧。當 objects 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 object 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，可以被正確地修正，但還有其他「指向此一 object」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「複製式收集器（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，可能只會產生少量的垃圾，甚至不產生垃圾。儘管如此，複製式收集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 會偵測是否沒有新的垃圾產生，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。對於一般用途而言，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 所依循的運作規則，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 objects。每當它找到一個存活的 object，便設定該 object 內的旗標，加以標示。此時 object 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 objects 會被釋放（譯註：因爲這些 objects 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 objects。

stop-and-copy 這一名稱，意味這種垃圾收集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收集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 垃圾收集器才會啓動，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：而非配合 object 大小的小塊記憶體）。如果你配置了大型 object，它會擁有專屬區塊。嚴謹的 *stop-and-copy* 作法，必須在你釋放舊有 object 之前，先將所有存活的 objects 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 可以在收集過程中使用已死區塊來複製 objects。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指涉（referenced），其世代計數會被否決。這種方式可以處理數量極大而生命短暫的暫時物件（temporary objects）。完整的清理動作會定期出現 — 大

型 objects 仍然不會被複製（只是其世代計數被否決而已）。內含小型 objects 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 objects 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。我們可以一句重要的話作總結：自省模式衍生出 *stop-and-copy* 和 *mark and sweep*。

JVM 可以加上許多額外的速度提昇技巧。載入器動作和 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 object 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。因此如果該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 method 之內，Java 會給你編譯期錯誤訊息，以貫徹它的保證。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 data member 隸屬基本型別，情況就稍有不同。因爲所有 methods 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值也很危險，所以每個隸屬基本型別的 class data member 都保證一定有初值。下面的例子顯示這些初值：

```
//: 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();
    */
  }
} ///:~
```

程式輸出結果如下：

```
Data type      Initial value
boolean        false
char           [ ] 0
byte           0
short          0
int            0
long           0
float          0.0
double         0.0
```

char 之值爲零，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 而卻沒有爲它設定初值（指向某個 object），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

如果你想爲某個變數設定初值，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非基本型別的 objects 設初值。如果 **Depth** 是個 class，你可以在加入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 method 來提供初值：

```
class CInit {
  int i = f();
  //...
}
```

當然，method 可以擁有引數，但這些引數不可以是尚未初始化的其它 class members。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。因爲它們和初始化順序有關，與程式編譯路線不符。

此種初始化方式極簡單又直覺，不過有個限制：每個 **Measurement** object 都有相同的初值。有時候這是你所要的，有時候你需要更彈性的方法。

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 methods 並執行某些動作以決定初值。不過，使用這種方式時，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 methods 定義之間。但所有變數一定會在任何一個 method（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 objects 身上。請參考《*Thinking in C++*，第二版》（內含於本書所附光碟片，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** 之中刻意將幾個 **Tag** objects 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 methods 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 object 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** — 除非你產生新的 object，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，不論產生多少個 objects，**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。下面這個例子清楚突顯出這一問題：

```
//: 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 定義中，產生 **Bowl static** members。請記住，**Cupboard** 會在 **static** 定義之前先產生 non-**static** 的 **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** object，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** object 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** object 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** objects。這一現象可從輸出結果觀察得到。

爲物件生成過程做出一份摘要整理，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 object 首次被產生，或是當 class **Dog** 的 **static** method 或 **static** field 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，這就是產生一個 **Class** object），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** object 首次被載入時。

3. 當你 **new Dog()**，**Dog** object 的建構過程會先爲 **Dog** object 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** object 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 method，但其實是在關鍵字 **static** 之後緊接著 method 主體。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 object 或首次存取該 class 的 **static** member（即使你從未產生過該 class object）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** object **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，因爲 **static** 初始化動作只會執行一次。

Non-static 的實體（instance）初始化動作

Java 也爲 object 內的 non-**static** 變數的初始化行爲提供了類似語法。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();
  }
} ///:~
```

在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*（無名的內層類別，詳見第八章）的初始化動作，此語法不可或缺。

Array（陣列）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 object。Java 也有 array，也支援 array 初始化功能。

array 其實不過是一連串 objects 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陣列有多大。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。儲存空間的配置（等同於以 **new** 來進行）在這種情況下由編譯器負責。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 objects 或基本資料）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。但是 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礎型別的 array（注意，**new** 無法產生 non-array 基礎型別資料）：

```
//: 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()** method 隨機選取而來，所以 array 的生成毫無疑問發生於執行時期。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「非基礎型別」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 objects）。以 **Integer** wrapper type（外包型別）爲例，它是個 class 而非基礎型別：

```
//: 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** 產生出 array 之後：

```
Integer[] a = new Integer[pRand(20)];
```

這還是個 references array。只有在「生成新的 **Integer** object 以初始化 reference」之後，初始化動作才算完成：

```
a[i] = new Integer(pRand(500));
```

如果你忘了產生 object，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** object 的形成。你看到了，指向 **Integer** object 的那個 reference 被自動轉型，變成 object 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（這個功能對冗長的初值列的維護頗有幫助。）

array 初始化的第二種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 method，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 objects 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，你將學到如何發覺此類 objects 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 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 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，其第一維元素的長度由亂數決定，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礎型別的 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礎型別的 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。

摧毀

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：編程上有極大比例的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 object），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 objects 必須被明確加以摧毀。但由於 Java 的垃圾收集器會自動釋放無用的 objects 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 method。在那些不需要解構式的場合，Java 垃圾收集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。某些垃圾收集器甚至可以清理其他如繪圖資源和 file handles 之類的資源。不過，垃圾收集器也增加了執行時期的成本。付出的代價很難清楚描述，因爲本書撰寫之際，整個 Java 直譯器到處充滿著緩慢因子。如果這點能夠改變，我們

便有能力判斷垃圾收集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」。（問題之一便是垃圾收集器具有不可預測的特性）

由於所有的 objects 都保證會被建構，所以和建構式有關的事物，實際上更多過本章所提出的說明。具體而言，當你運用複合技術（*composition*）或繼承技術（*inheritance*）來產生新的 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. 請撰寫一個帶有預設建構式（default constructor，亦即不接收任何引數）的 class，並在其中列印出某些訊息。請爲此 class 產生一些 objects。
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 objects 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 objects，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()** method。此一 method 應根據不同的基礎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 methods 都有兩個引數（兩種不同型別），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 object，驗證預設建構式是否真的被自動合成。

9. 撰寫具有兩個 methods 的 class，在第一個 method 呼叫第二個 method 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()** method 的 class，並在此 method 中列印訊息。請在 **main()** 中針對該 class 產生一個 object。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：object 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** method 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 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 method，能夠產生二維的 **double** array 並加以初始化。array 的容量由 method 的引數決定，其初值必須落在 method 的引數所指定的上下限之間。撰寫第二個 method，印出第一個 method 所產生的 array。試著在 **main()** 中透過這兩個 methods 產生不同容量的 array，並列印其內容。

20. 重覆題 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。重覆多次，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: 隱藏實作細節

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 libraries 來說尤其重要。Library 必須搏得其使用者（即客端程式員，*client programmer*）的信賴。程式員必須確認，即使 library 釋出了新版本，他們亦無需改寫程式碼。從另一方面說，library 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 library 內的 classes 時，library 開發者必須遵守「不刪去任何既有 methods」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 data member 而言，library 開發者如何才能夠知道，究竟哪些 data members 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 methods 來說，情形亦同。如果 library 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 members 的任何更動，都可能造成客端程式碼無法正常運作。library 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 library 開發者得以指明哪些 members 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 library 設計者，你會希望儘可能讓每個 member 都是 **private**，而且僅對外公開你希望客端程式員使用的 methods。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。本章結束之際，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，components library（組件程式庫）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 library unit，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 位於同一個 package 或另一個 package」的影響。因此，本章一開始，你會先學習如何將 library 元件置入 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 library，例如：

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

你取用的便是所謂的 package。這種寫法會將標準的 Java utility library（也就是 **java.util**）整個引入。舉個例子，class **ArrayList** 位於 **java.util** 中，所以你可以選擇指定其全名 **java.util.ArrayList**（如此便毋需動用上述的 **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 members 的名稱皆被相互隔絕。class **A** 的 method **f()**，其名稱不會和 class **B** 中具有相同標記式（signature，亦即引數列）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 被用於網際網路，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 必須不受網際網路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，因而沒有遭遇 package 的命名困擾。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 程式能與同台機器上的其他 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 **.java** 副檔名）。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 library，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爲 **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。

假設上例的檔案名爲 **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 **import** 關鍵字，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）

import 關鍵字可以造成比較簡潔的效果：

```
import mypackage.*;
// . . .
MyClass m = new MyClass();
```

身爲 library 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網際網路，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 開發者的網際網路域名（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。

上述技巧的第二個部份，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 object、或首次存取 class **static** member 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。

我以我的域名 **bruceeckel.com** 爲例，做更清楚的解說。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 library，所以我更進一步細分此一名稱，獲得的 package 名稱爲：

```
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，這兩個 classes 以及 **Vector**、**List** 之中欲被使用的 methods，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 libraries 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，編譯器會產生錯誤訊息，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.*** 。

設計 library

有了以上認識，現在你可以開發自己專屬的 tool libraries 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()** 的別名，減少打字負擔。這個 class 可做爲名爲 **tools** 之 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 objects 都可輕易地被強迫轉成 **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，而毋需更動程式碼。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，所以應該不需要此一功能。

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，在出貨產品中加以關閉。Allen Holub（*www.holub.com*）提出了以 packages 來模擬條件編譯的想法。他根據此一想法在 Java 裡頭產生 C 語言中極爲有用的 *assertion* 機制。憑藉著此一機制，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 檢驗動件，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，該工具稱爲「異常處理（*exception handling*）」。此刻，**perr()** method 便已足夠我們用的了。

上述 class 的輸出結果會藉由「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 assertions，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");
  }
} ///:~
```

改變匯入的 **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 時的一些問題

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 — 即使它們就位於同一目錄中。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 member 定義之前，無論此 member 究竟爲 field 或 method。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。它的意思是，同一 package 內的其他所有 classes，都可以存取 friendly member，但對於 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 members 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。你可以說，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，你或許會想排除掉

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 members 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 **Bo**b 的朋友！」然後就希望看到 **Bob** 的 **protected**、friendly、**private** members。如果想要對外授予某個 member 的存取權限，唯一的方法是：

1. 宣告該 member 爲 **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 member 成爲 friendly，並將其他 classes 置於同一個 package 內，於是其他 classes 便可存取該 member。
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** member（但不含括 **private** members），就和存取 **public** member 一樣。只有當兩個 classes 位於同一個 package 時，才能存取對方的 friendly members。不過此刻毋需擔憂此事。
4. 提供「存取式（accessor）/變異式（mutator）」（也被稱爲 "get/set" methods），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 member 宣告可爲每個人所用」，特別是對於此一 library 的使用者（客端程式員）。假設你定義了名爲 **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**（代表本書的第 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。千萬別誤以爲 Java 一定會將目前所在目錄做爲搜尋起點之一。如果你沒來將 '.'（譯註：也就是目前所在目錄）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** object，因爲 **Cookie** 的建構式爲 **public** 而其本身亦爲 **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()** method，因爲 **bite()** 的存取權限爲 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** object 並呼叫其 **f()** method！（請注意，若想編譯這些檔案，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餘檔案來說，它們都是 friendly。

private: 不要碰我！

關鍵字 **private** 表示「除了當事人（member）所在的 class，沒有任何人可以存取這個 member」。即使是同一個 package 內的其他 classes，也無法存取你的 **private** members。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 member，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 member。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 members，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 object；你得呼叫 **makeASundae()** method，讓它來爲你服務[2]。

只要你確信某些 methods 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 method 宣告爲 **private**，可保證你自己仍握有決定權。

---

[2] 此處還會產生其他效應。因爲我們僅定義了預設建構式（default ctor），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）。

對於 class 內的 **private** field 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 fields 都成爲 **private**。不過，在 class 中令某個 object reference 爲 **private**，並不代表其他 objects 無法擁有該 object 的 **public** reference。請參考附錄 A 中關於 aliasing（別名）的種種討論。

protected: 某種友善

想要了解 **protected** 存取權限飾詞，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 members 加到被我們稱爲 base class（基礎類別）的既有 class 中，而無需碰觸既有的 base class。你也可以改變 base class 既有的 members 的行爲。如果想繼承某個既有的 class，你可以讓新的 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，並且繼承了另一個 package 中的 class，那麼你就只能存取原先 package 中的 **public** members（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 members 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）存取某個特定的 member，但不希望所有 classes 都有此權力。而這正是 **protected** 的用途。以先前出現的那個 **Cookie.java** 爲例，以下的 class 將無法存取其 friendly member：

```
//: 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** 中存在 method **bite()**，那麼它也會存在於繼承自 **Cookie** 的所有 classes 中。但由於 **bite()** 是另一個 package 中的 friendly method，所以無法被這些 derived classes 取用。你當然可以將它宣告爲 **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爲是一種「實作細目的隱藏（*implementation hiding*）」。將 class 內的 data 和 methods 包裝起來，結合實作細目隱藏，即是所謂的封裝（*encapsulation*）[3]。其結果就是一個兼具特性與行爲的資料型別（data type）。

[3] 即是單單只是實作細目的隱藏，人們也常稱之爲封裝。

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 － 將介面與實作分離。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** methods），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**（例如 friendly、**protected**、**private**）事物。

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「某一類型的 objects」，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 object 都具備同樣的特性和行爲。所謂 class，就是所有同型 objects 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 data 和 methods」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** members 置於 class 起始處，其後再接著 **protected**、friendly、private members。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（此即 **public** members，因爲它們可於檔案之外被取用），而在看到 non-**public** members 時停下來，因爲已經到了內部實作細節部份：

```
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，因爲介面與實作仍舊混在一起。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 之中。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（亦即顯示出可用之 members）的一種工具。在你閱讀本書的時候，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「library 中的哪些 classes 可以被 library 使用者所用」。如果你希望某個 class 可以被客端程式員所用，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 objects。

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 library 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）必須恰與其在的編譯單元（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，但編譯單元中的確可以不含任何 **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，你可能會徹底改變作法並完全捨棄舊版本，以全新版本替代。如果想擁有如此彈性，你得確保沒有任何客端程式員倚賴 **mylib** 內的特定實作細目。欲達到此一目的，只要拿掉 class 的 **public** 飾詞，它就成爲了 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，請將所有建構式宣告爲 **private**，這麼一來便可阻止任何人產生其 object，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** member 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 methods 皆傳回 **void** 或基礎型別，因此以下定義乍看之下令人有點困惑：

```
public static Soup access() {
  return ps1;
}
```

上述 method 名稱（**access**）之前的字，指出該 method 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 method 所回傳的，即是一個 reference，代表一個 class **Soup** object。

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 的 object。請千萬記住，如果你沒有自行撰寫至少一個建構式，會有一個預設建構式（default constructor，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預設建構式，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 object。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** method 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** objects 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）。這個樣式人稱 "singleton"，因爲它讓整個程式面對某個 class 時只能產生其唯一一個 object。class **Soup** 的 object 是被 **Soup** 的 **static private** member 產生出來的，所以恰恰只能有一份。而且，除了透過 **public** method **access()** 加以存取，別無它法。

如前所述，如果你未指定某個 class 的存取權限飾詞，預設便是 friendly。這表示同一個 package 內的其他 classes 能夠生成該 class 的 objects，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 的某個 **static** member 爲 **public**，那麼客端程式員即使無法生成該 class 的 object，仍然能夠存取這個 **static** member。

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 library 時，會建立起與使用者（亦即客端程式員，也就是將你的 library 結合至應用程式，或藉以開發更大型 library 的人）之間的關係。

如果缺乏規範，客端程式員可以對 class members 進行任何他們想進行的動作 — 即使你可能不希望他們直接操作這些 members。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 libraries。首先介紹如何將一組 classes 包裝於 library 之中，然後介紹如何控制 class members 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。因爲 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以輕易加以規避。

我們之所以要控制 members 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一些東西；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 methods 與 fields 宣告爲 **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 library 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 library 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「倚賴 class 底層實作細節」的使用者。身爲 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，因爲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。因爲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 methods — 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，在其產生 **ArrayList** object，但不明確地匯入 **java.util.***。
2. 將標題爲「package：程式庫單元（library unit）」一節中與 **mypackage** 有關的程式碼片段，改寫爲一組可編譯、可執行的 Java 檔案。
3. 將標題爲「衝突」一節中的程式碼片段，改寫爲完整程式，並檢查實際發生的命名衝突。
4. 將本章所定義的 **P** class 泛化：加入 **rint()** 和 **rintln()** 的所有重載版本，使其足以處理所有不同的基本 Java 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 data members 和 method members 的 class。爲它產生一個 object 並進行觀察，當你嘗試取用所有 class members 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 之中。

7. 撰寫一個 class，令它具備 **protected** data。並在同一個檔案中撰寫第二個 class，爲此 class 提供 method，使它操作第一個 class 的 **protected** data。

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 library，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 methods 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** objects。請你限制客端程式員，使他們無法自行產生 **Connection** objects，只能經由 **ConnectionManager** 的 **static** method 來獲得這些 objects。當 **ConnectionManager** 之中不再存有任何 objects 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();
   }
} ///:~
```

請你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、然後加以改變」的舊有模式。

C 之類的程序性語言（procedural language）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 objects。這種方法稱爲「複合（*composition*）」，或稱組合，因爲新的 class 是由既有 classes 的 objects 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。

對複合和繼承而言，其語法與行爲大多類似（這麼做饒富意義，因爲二者都是從既有型別產生出新型別）。本章之中，你會學到這兩種「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 object，它必須能夠儲存多個 **String** objects、兩三個基礎型別資料、以及另一個 class object。你可以直接定義基礎型別

資料，但是對非基礎型別的 objects 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 methods 中，有一個很不同尋常：**toString()**。稍後你會學到，每個非基礎型別的 objects 都具備 **toString()** method，當編譯器希望得到一個 **String**，而你手上卻只有這些 objects 的情況下，這個 method 便會被喚起。所以下列算式：

```
System.out.println("source = " + source);
```

編譯器發現你企圖將 **String** object（"**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**！」完成這個動作之後，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()** method 即可。

乍見之下你可能會假設（喔是的，Java 應該表現出安全謹慎的一面）編譯器自動爲上述程式碼中的每一個 references 產生相應的 objects；例如它會呼叫 **WaterSource** 的預設建構式，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 之中隸屬於基礎型別的 fields，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 methods，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 產生預設的 object」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 object 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 object 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 object 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，在 **Bath** 建構式中，述句的執行會在任何初始化動作之前發生。如果你未在定義處進行初始化動作，無法保證你在發送訊息給 object references 之前能夠進行任何初始化動作 — 除了執行期異常。

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 fields 在它們被使用之際都已被妥善地初始化了。

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。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「新的 class 和舊的 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 data members 和 methods。

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** 的 **append()** method 中我以 += 運算子將 **String** object 接續於 **s** 之後。+= 運算子是 Java 設計者加以重載後用以處理 **String**s 的運算子之一（另一個是 "+"）。

第二，**Cleanser** 和 **Detergent** 都含有 **main()** method。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 之內。不過，如果某個程式中存有許多 classes，僅有命令行所調用的那個 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，你的命令行可以是 **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()**明白呼叫了 **Cleanser.main()**，並將命令行引數原封不動地傳過去（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 methods 都是 **public**。請千萬記得，如果你未指定 member 的存取權限飾詞，預設便爲 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飾詞，同一個 package 中的所有 classes 皆能使用這些 methods。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** members。所以，爲了繼承著想，一般原則是將所有 fields 宣告爲 **private**，將所有 methods 宣告爲 **public**（**protected** members 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的準則。

請注意，**Cleanser** 的介面含括了一組 methods：**append()**、**dilute()**、**apply()**、**scrub()**、**print()**。由於 **Detergent** 衍生自（*derived from*）**Cleanser**（透過關鍵字 **extends**），所以它會自動從 **Cleanser** 的介面

中取得所有 methods。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，但並非最主要的部份。

正如 **scrub()** 所顯示，你可以修改定義於 base class 中的 methods。在這個例子中，你可能會想要在新版本中呼叫 base class methods。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()** method。

實施繼承時，你不一定非得使用 base class methods 不可。你也可以將新的 methods 加至 derived class，就像將 method 加入一般 class 沒有兩樣：只要加以定義即可。**foam()** method 便是一個例子。

在 **Detergent.main()** 中你可以看到，除了 **Detergent** methods（例如 **foam()**）之外，你也可以呼叫 **Cleanser** 的所有可用的 methods。

base class 的初始化

由於現在有兩個 classes 牽扯進來了：base class 和 derived class，而不單只是一個，所以當你試著想像 derived class 所產生的 object 時，可能會感到難以理解。外界看來，它像是一個具有「和 base class 同樣介面」的新 class，也許還多了些額外的 methods 和 fields。但繼承不單只是複製 base class 的介面而已。當你產生 derived class object 時，其中會包含 base class *subobject*（子物件）。這個 subobject 就和你另外產生的 base class object 一模一樣。外界看來，base class subobject 被包裝於 derived class object 之內。

當然，將 base class subobject 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與權力。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。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();
  }
} ///:~
```

上述程式的輸出說明了自動呼叫行爲：

```
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預設建構式，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 都具備預設建構式，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 並不具備預設建構式，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）。

Catching base constructor exceptions

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。這有時候挺不方便的。

組合（composition）與繼承（inheritance）

同時使用複合與繼承，是極爲常見的事。以下例子同時使用了兩種技術，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");
  }
}
```

```
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，而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 member objects 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 object 被摧毀時自動被呼叫的一個 method。這或許是因爲 Java 裡頭的「object 摧毀方式」很簡單，只要忘掉 objects、讓垃圾收集器在必要時候回收其所佔記憶體，就可以了，無需明白加以摧毀。

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，你無法得知垃圾收集器被喚起的時機，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 method 來進行此事，並讓客端程式員確知他們得呼叫此一 method 來完成清理工作。首要之務 — 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**，因爲它暗中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 method。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 object 生命期中呼叫的每個 methods，其實都可能做些需要清理的事。每個 class 都有專屬的 **cleanup()** method，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() 之中出現了兩個前所未見的關鍵字：**try** 和 **finally**。本書直到第 10 章才會正式將它們介紹給你。關鍵字 **try** 表示，接下來的區段（以一組大括號括起的範圍）即所謂的守護區（*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 method 中，你得留意「呼叫 base class 和 member object 的 cleanup methods」的次序，以防範「某個 subobject 與另一個 subobject 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 的所有清理動作（其次序和生成次序相反），這得「以 base class 產生的元素」都仍存活才行。然後，就像此處所示範的，呼叫 base class 的 cleanup method。

許多時候，「清理（cleanup）」議題不是問題；你只要讓垃圾收集器做事就好了。但是當你必須自行處理時，你得更加努力並且小心。

垃圾收集順序

一旦事情和垃圾收集起關聯，就不再有太多你可以信賴的事。垃圾收集器可能永遠都不會被喚起；即使它被喚起，也有可能以任何它想要的次序來回收 objects。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 methods，不要倚賴 **finalize()**（正如第 4 章所說，你可以強迫 Java 呼叫所有的 finalizers）。

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 method 名稱，那麼在 derived class 中重新定義此一 method，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 method，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 methods，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）。

複合和繼承之間的抉擇

Choosing composition vs. inheritance

複合與繼承，都讓你可以將 subobjects 置於新的 class 之中。你可能會納悶，二者之間究竟有何差異，而且何時應該選用哪一種技術呢？

當你想要在新的 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 object，使你得以在新的 class 中以它來實現你想要的功能。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 object 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 classes 的 objects。

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 member objects 宣告爲 **public**）是有意義的。如果各個 member objects 分別完成了實作細目隱藏，這種作法很安全。當使用者知道你組合了一堆零組件，他們便更能夠輕易了解其介面。**car** object 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，也爲問題分析的一部份（而不僅是底層設計的一部份），所以，將 members 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 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。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** members 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 data members 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** methods 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。藉著繼承既有的、基礎的 classes，並且加上新的 data members 和 methods（並重新定義既有的 methods），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 methods 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的、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 的處理，應該透過許多不同類型的 objects 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 methods」。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：「新的 class 是既有的 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 methods」，所以你可以發送給 base class 的任何訊息，也都可以發送給 derived class。如果 **Instrument** class 擁有 **play()** method，那麼 **Wind** 也會有。這表示我們可以很精確地說，**Wind** object 也是一種 **Instrument**。以下例子說明了編譯器如何支援此種表示式：

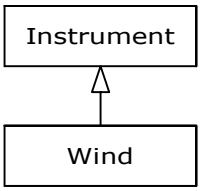
```
//: c06:Wind.java
// Inheritance & upcasting.
import java.util.*;

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()** method，它接受一個 **Instrument** reference。然而當 **Wind.main()** 呼叫 **tune()** method 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 method，竟然可以接受另一種型別。在你明白「**Wind** object 其實也是 **Instrument** object」之前，你可能會覺得這真是件怪事。此例之中，沒有什麼 methods 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 轉爲 **Instrument** reference」的動作被稱爲「向上轉型（*upcasting*）」。

爲什麼叫它向上轉型（Why "upcasting"）？

這個詞彙有其歷史背景，而且依據繪製繼承圖時的傳統方式：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** 的繼承圖爲：



從 derived class 移至 base class，在繼承圖中是向上移動，所以通常稱爲「向上轉型」。向上轉型一定安全，因爲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 可能包含多於 base class 的 methods，但它至少一定包含 base class methods。向上轉型過程中，對 class 介面造成的唯一效應，是 methods 的「遺失（lose）」而非「獲得（gain）」。這也就是爲什麼在「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。

你也可以執行與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 章所討論的課題有關的難題。

複合（Composition）vs. 繼承（inheritance），再探

物件導向程式設計中，產生和運用程式碼的最可能形式，便是將 data 和 methods 包裝起來成爲 class，然後運用其 objects。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。所以雖然繼承在 OOP 的學習過程中獲得許多強調，但並不代表你應該處處使用它。

相反的，你應該謹慎使用它：只有在很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 **final** 關鍵字有所誤解。一般來說，它的意思是：「這是不能被改變的」。基於設計和效率兩大理由，你可能希望阻止改變。由於這兩個理由十分不同，可能造成對 **final** 關鍵字的誤用。

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. 可以是在執行期（*run-time*）被初始化，而你卻不想再加改變的值。

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，減少執行期負擔。在 Java 中，此類常數必須屬於基礎型別，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。

如果某個 field 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 而不是用於基礎型別時，其實際意義可能有點令人困惑。用於基礎型別時，**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，代表某個 object 之後，它便再也不能改而指向另一個 object。但此時 object 本身的值卻是可以更動的；Java 並未提供「讓任何 object 保持不變」的機制（不過你還是可以撰寫你自己的 class，產生令 objects 保持不變的效果）。此一限制對同樣身爲 objects 的 array 而言，也是成立的。

下面是個例子，用來示範 **final** fields：

```
//: 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** 基礎型別，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** 的定義方式則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**，用以強調它只有一份；定義爲 **final**，用以宣告它是個常數。請注意，帶有常數初值的 **final static** 基礎型別，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，在執行期使用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** object 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，不會在每次產生新的 object 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()**中所觀察到的，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 object，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** 似乎比將基礎型別宣告爲 **final** 來的不實用多了。

Blank finals

Java 允許產生所謂「留白的 finals」（*blank final*），也就是允許我們將 fields 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** field 便可以指向不同的 object，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** field 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 method 中更改該引數（一個 reference）所指之物：

```
//: 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 指派給宣告爲 **final** 的引數，就像面對 non-**final** 引數一樣。是的，這麼做並不會帶來編譯的困擾。

f() 和 **g()** 這兩個 methods 說明了「當基礎型別引數爲 **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 的原因有二。第一，鎖住這個 method，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 method 的行爲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 method 宣告爲 **final**，等於允許編譯器將所有對此 method 的呼叫動作轉化爲 *inline*（行內）呼叫。當編譯器看到一個 **final** method 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「method 呼叫機制」：將引數送入 stack、跳至 method 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 method 本體取代那個呼叫動作。這麼做能夠降低 method 呼叫動作所引發的額外負擔。當然，如果 method 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，你獲得的效率利益將因爲「花費在 method 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** method 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 method 真的體積很小或是你真的不希望它被覆寫時，才將它宣告爲 **final**。

final 和 private

class 中的所有 **private** methods 自然而然會是 **final**。因爲你無法取用 **private** method，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

method）。你可以將 **final** 飾詞加至 **private** method 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** method（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「method 屬於 base class 介面」時。也就是說，你必須能夠將某個 object 向上轉型至其基礎型別，並呼叫同一個（同名）method（下一章將對此點做更進一步的釐清）。如果某個 method 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" method，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** method 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 宣告爲 **final**（將關鍵字 **final** 置於其定義之前），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**，data members 既可以是 **final**，也可以不是，依你的意志而定。**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 methods 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 method 宣告爲 **final**，編譯器一樣具有效率上的選擇權。

你可以將 **final** 飾詞加於 **final** class 內的 method 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 methods 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 methods。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 method 定義爲 **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。因爲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 methods 都未被宣告爲 **final**，它可能會更爲有用。你可能想要繼承如此有用的 base class 並加以覆寫。這種想法很容易理解。但是不知怎的，其設計者卻認爲這麼做不妥當。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** 中的許多重要 methods，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。這種情況引導人們相信「猜測最佳化動作應該發生於何處的那些程式員，永遠都是差勁的程式員」這句話。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 container library 以 **ArrayList** 汰換了 **Vector**。**ArrayList** 的行爲合理多了，不幸的是，許多新開發的程式仍然使用那些老舊的 container library。

Hashtable 亦頗值得一書。它是另一個標準程式庫中的 class，但未具備任何 **final** methods。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 methods 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 container library 已經以 **HashMap** 汰換了 **Hashtable**。

初始化以及 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 **static**s 的初始化順序不會引發問題。以 C++爲例，如果某個 **static** 預期另一個 **static**「在初始化之前，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 objects，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 object 被建構之時，也可能是在某個 **static** field 或 **static** method 被存取時。

「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** objects 和 **static** code block（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）被初始化。當然，**static**s 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** method），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 object，這個動作都會發生。請試著將其 object 生成動作註解掉，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 members 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 objects 了。首先，object 內的所有基礎型別都會被設予預設值，object reference 則被設爲 **null** — 也就是說，只要將 object 記憶體都設爲二進位零值即可。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」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。

儘管物件導向程式設計極爲強調「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，你可以在執行時期改變 member objects 的實際型別和行爲，如此一來便可以在執行時期改變複合而成的（composed）objects 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. 請撰寫兩個帶有預設建構式（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 class **B** object 做爲 member。請不要爲 **C** 提供任何建構式。現在，產生一個 class **C** object，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，而非預設建構式（default ctor）。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 field 定義爲第一個 class 的 object。請使用緩式初始化（lazy initialization）來產生這個 object 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 method。

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預設建構式，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，令它僅具有一個非預設建構式。再撰寫一個 derived class，令它同時具備預設建構式和非預設建構式。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**，並令它含有各個 "component"。所有 classes 都應該具備能夠印出 class 相關訊息的一個預設建構式。

11. 修改上題的程式，令每個 class 僅有非預設建構式。

12. 將 **cleanup()** methods 適當加至練習 11 中所有 classes。

13. 撰寫某個 class，令它擁有一個重載三次的 method。爲它衍生一個新的 class，並爲上述 method 加入一份新的重載定義。證明這四個 methods 在 derived class 中都可用。

14. 在 **Car.java** 中，請將 **service()** method 加至 class **Engine**，並在 **main()** 中呼叫它。

15. 撰寫位於 package 中的某個 class，並令它含有一個 **protected** method。在此 package 之外，試著呼叫該 **protected** method，並解釋其結果。接著，繼承剛才那個 class，並在 derived class 的某個 method 中呼叫上述的 **protected** method。

16. 撰寫一個 **Amphibian** class，並令 **Frog** 繼承之。請將適當的 methods 置於 base class 中。並在 **main()** 中產生 **Frog** object，且向上轉型至 **Amphibian**。示範所有的 methods 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 method 定義（以同樣的 method 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** field 和 **final** field，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 method（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** method。衍生一個新的 class 並覆寫該 method。

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** member 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「將特性（*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 object 以其本身型別視之，亦能以其基礎型別視之。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（不同的）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。相似型別之間的區別可藉由「methods 的行爲差異」來展現，而這些 methods 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，object 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，毋需任何轉型動作。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。

忘掉 object 的型別

你可能覺得這個程式頗爲奇怪。爲什麼有人要刻意把 object 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 methods。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 method，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

methods，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 method，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。

疑問

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()** method：

```
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 繫結方式

所謂「繫結（*binding*）」，就是建立 method call 和 method body 的關聯。如果繫結動作發生於程式執行前（由編譯器和連結器執行），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 method。

解決方法便是透過所謂的後期繫結（*late binding*）：繫結動作在執行期根據 object 的型別而進行。後期繫結也被稱爲執行期繫結（*run-time binding*）或動態繫結（*dynamic binding*）。程式語言欲實作出後期繫結，必須具備「得以在執行期判知 object 型別」並「呼叫其相應之 methods」的機制。也就是說，編譯器仍然不知道 object 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 objects 之中。

Java 的所有 methods，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。

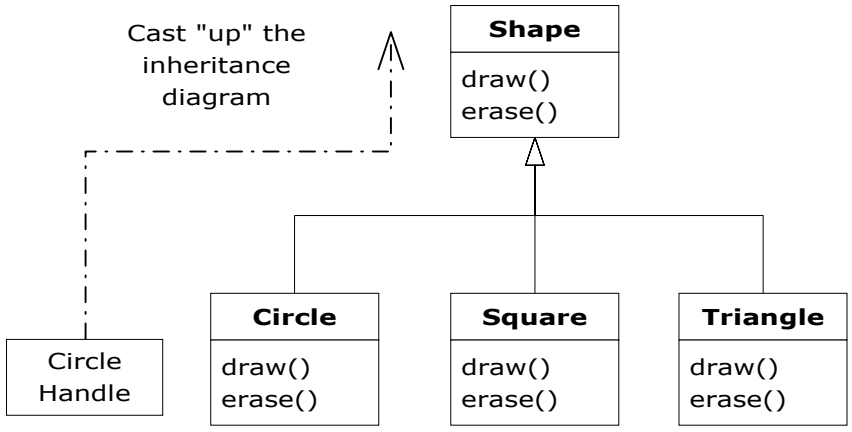
爲什麼會想將某個 method 宣告爲 **final** 呢？一如前一章所述，這是爲了防止其他人覆寫該 method。但或許更重要的是，這麼做可以「關閉」動態繫結。或者說，這麼做便是告訴編譯器：動態繫結是不需要的。於是編譯器便得以爲 **final** method call 產生效率較佳的程式碼。不過大多數時候這麼做並不會爲你的程式帶來整體效能提昇。所以最好是基於設計上的考量來決定是否使用 **final**，而不要企圖藉由它來改善效能。

再論正確的行為

一旦你知道所有 Java methods 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 object，讓該 object 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。我們可以簡單地說「圓（circle）是一種形狀（shape）」，而且也容易理解。這便是這個例子之所以好的原因。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** object，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 method（它被覆寫於 derived class）：

```
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()**。

下面這個例子有了一點變化：

```
//: 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** 含有一個名爲 **randShape()** 的 **static** method。每次呼叫這個 method，它會產生（並回傳）一個 reference，指向隨機挑選的一個 **Shape** object。請注意，其中每一個 **return** 述句，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 method，你永遠無法察知你所獲得的 object 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()
```

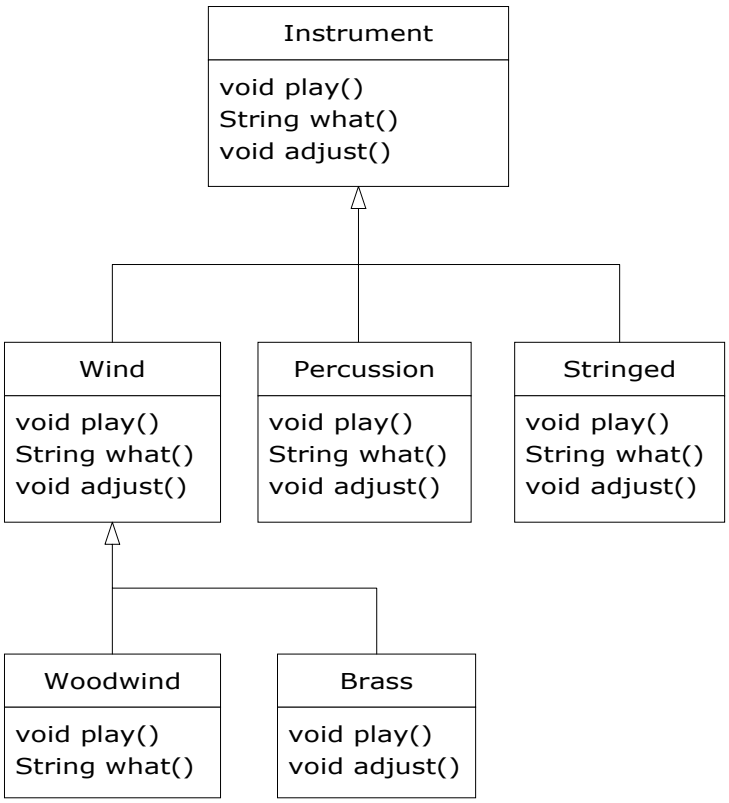
```
Square.draw()
```

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()** method。在妥善設計的 OOP 程式中，大多數或甚至全部的 methods 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 methods，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 methods 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()** method，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 methods，但 **tune()** 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 {
  public void play() {
    System.out.println("Instrument.play()");
  }
  public String what() {
    return "Instrument";
  }
```

```
  public 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 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 methods 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 之內，形同自動向上轉型至 **Instrument**。

我們發現，**tune()** method 很幸福地完全不知道它週遭程式碼所發生的變動，而依舊正常運作。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()** method 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 method，而是加以重載（*overloaded*）。編譯器允許你進行 method 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要的。以下便是例子：

```
//: 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()** method 接受 **int**，並以 **NoteX** 做爲識別字。是的，即使 **NoteX** 是個 class 名稱，它仍舊可被用來做爲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** 做爲識別字（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 method 時有了小小的打字失誤。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。

在 **tune()** 中，發送 **play()** 訊息給 **InstrumentX i**，並以 **NoteX** members 之一（**MIDDLE_C**）做爲引數。由於傳入的 **NoteX** member 爲 **int**，所以呼叫的是本例重載後的 **play()** method 的 **int** 版本 — 因爲沒有 **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 與 Abstract methods

前述的樂器實例中，base class 內的所有 methods 皆只有掛名性質。如果這些 methods 被呼叫了，可能會引發一些問題。這是因爲，**Instrument** 的存在目的僅止於爲它的所有 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** 稱作「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 methods，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 method 名稱和 base class method 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 objects 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** object 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 methods 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 method，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 的 class，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 object 時，編譯器如何反應呢？唔，爲 abstract class 產生任何 object 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。

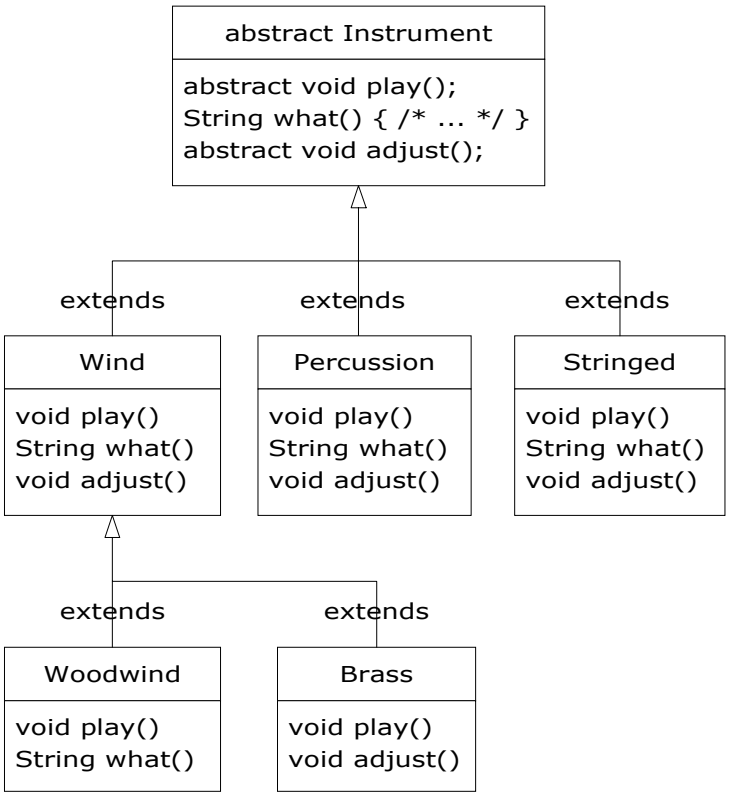
如果你繼承一個 abstract class，並且希望爲新型別產生 objects，那麼你得爲 base class 中的所有 abstract methods 都提供相應的 method 定義。如

[1]對 C++程式員而言，abstract method 相當於 C++ 的純虛擬函式（*pure virtual function*）。

果沒有這麼做（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 methods 會變成 abstract，因爲將某個 class 宣告爲 **abstract**，並不強迫你得將所有 methods 都宣告爲 **abstract**。以下是可能的樣子：



以下便是運用了 **abstract** classes 和 **abstract** methods 之後的管弦樂器修定版：

```
//: 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）與多型（polymorphism）

一如以往，建構式異於其他 methods。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 members，無法存取 base class 的 members（這些 members 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 object 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 的預設建構式便會被自動喚起。如果缺乏預設建構式，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預設建構式）。

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 member objects。當 **Sandwich** object 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 object 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 members 的宣告順序，呼叫各個 member 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 members。這意味當你置身於 derived class 時，你必須假設 base class 的所有 members 都是有效的。在一般 method 之內，建構動作早已完成，所以 object 內的所有成份（譯註：因複合而形成的東西）的所有 members 皆已建構完成。然而在建構式中，你得設法讓你所使用的 members 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 members 便已初始化完畢。在建構式中「確知所有 members 皆可被合法取用」這一理由，促使我們儘可能在 member objects（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 members 以及當前這個 object 的 member objects 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 member objects 的終止化（*finalizing*）問題。每個 member 都是獨立的 object，因此會被垃圾收集器回收與終止化。這和它恰巧是你的 class member 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）得在垃圾收集時進行，你就得在 derived class 中覆寫 **finalize()**。當你這麼做時，千萬別忘了呼叫 base class 的 **finalize()**，因爲如果不這樣，base class 的終止化動作（*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()**」。程式會依據命令行引數（command line argument）來設定旗標值，讓你觀察 base class 終止化進行與否的相關行爲。

在這個階層體系中，每個 class 都含有一個 **Characteristic** member object。你會發現，無論 base class 的終止式（finalizers）是否被呼叫，**Characteristic** member objects 都一定會被終止化。

每個被覆寫的 **finalize()** 至少得具有存取 **protected** members 的權限，因爲 class **Object** 的 **finalize()** method 爲 **protected**，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** 旗標會被設定，並產生一個 **Frog** object。記住，垃圾收集動作 — 更明確地說是終止化動作 — 可能不會發生於某個 object 身上。所以，爲了強迫讓垃圾收集動作發生，我們呼叫 **System.gc()** 來觸發垃圾收集動作的進行，並因此引發終止化動作。如果不進行 base class 終止化動作，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 member objects 則如你所預期地被終止化了）。如果你在命令行加入 "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 member objects 的終止化順序和生成順序相同，但技術上而言，objects 的終止化順序是未經規範的。不過，透過 base classes，你可以控制終止化順序。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 的終止化順序，然後才是 base class 的終止化順序。這是因爲 derived class 的終止化動作可能會呼叫某些 base class methods，而這些終止化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 object」的某個動態繫結的 method，會發生什麼事？如果這發生在一般 method 之中，你可以想像會發生什麼事 — 執行時期會解析這個動態繫結呼叫動作，因爲 object 並不知道它究竟屬於此一 method 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 method，會喚起該 method 被覆寫後的定義。然而其效應無法預期，甚至可能會遮蓋某些難以發現的程式臭蟲。

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 object 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 method。如果你在建構式中這麼做的話，你便會喚起某個 method，其中可能取用尚未被初始化的一些 members。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()** method 是 **abstract**，所以它的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 method，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，其值當時爲 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 object 的儲存空間會被初始化爲二進制零值（binary zero）。

2. 以先前所述方式，呼叫 base class 建構式。此時，覆寫後的 **draw()** method 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** 值爲零。

3. 以 members 宣告順序來呼叫各 member 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爲零值（不論對特定資料型別而言零值是否有意義），而不會是雜七雜八的內容。即使是「藉由複合手法而被置於 class 內」的 object references，其值都會是 **null**。所以如果你忘了爲該 reference 設定初值，便會在執行時期收到異常。所有其他 objects 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，其行爲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爲）。諸如此類的程式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，一條指導原則便是：「儘可能少做事便使 object 進入正確狀態。如果可以的話，別呼叫任何 methods」。建構式中唯一可以安全呼叫的 methods 便是「base class 中的 **final** methods」（這對 **private** methods 來說一樣成立，因爲它們天生就是 **final**）。此類 methods 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，因爲多型是如此巧妙。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」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，便得在編譯時期知道確切的型別。以下例子說明這一點：

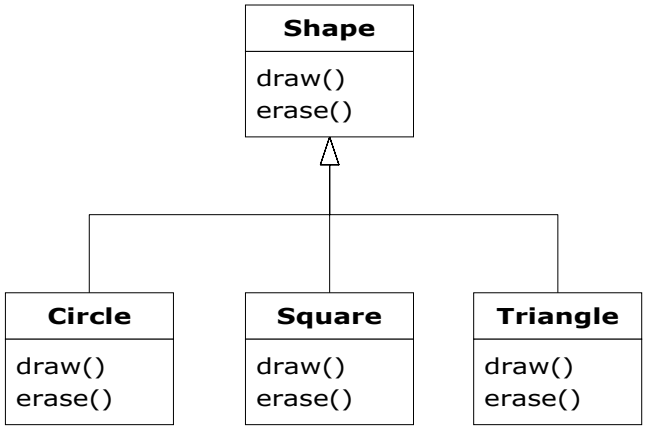
```
//: c07:Transmogrify.java
// Dynamically changing the behavior of
// an object via composition.

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** object 含有一個 **Actor** reference，其初值指向一個 **HappyActor** object。這表示 **go()** 會產生某種特殊行爲。由於 reference 可於執行時期被重新繫結至另一個不同的 object，所以我們可以將「指向 **SadActor** object」的 reference 置入 **a** 中，造成 **go()** 的行爲改變。於是我們獲得了執行時期的動態彈性。這種手法又被稱爲 *State Pattern*，詳見《*Thinking in Patterns with Java*》（此書可於 *www.BruceEckel.com* 下載）。相較之下，你無法於執行時期決定繼承對象，你一定得在編譯時期決定。

下面是個一般準則：「以繼承來表達行爲上的差異，以欄位（fields）來表達狀態上的變化」。上述例子同時使用了二者：兩個不同的 classes 被衍生出來，表達 **act()** method 的行爲差異，**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爲變化。

純粹繼承（Pure inheritance）vs. 擴充（extension）

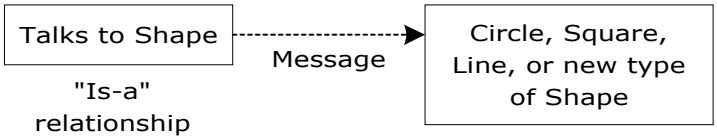
學習繼承時，最聰明的方法似乎是採用「純粹」繼承來建立整個繼承體系。也就是說，只有 base class（或謂 **interface**）所建立的 methods，才被 derived class 加以覆寫，如下所示：



這種作法可被視爲純粹的「是一種（is-a）」關係，因爲 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

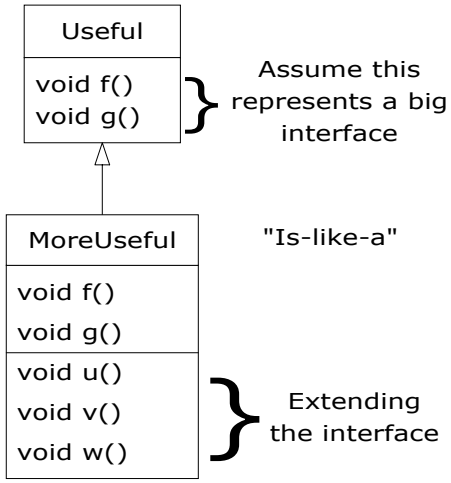
同的介面，而且一模一樣。如果你採用上述圖示，那麼 derived classes 除了「base class 介面」之外一無長物。

這種方式可被視爲「純取代（*pure substitution*）」，因爲 derived class objects 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。

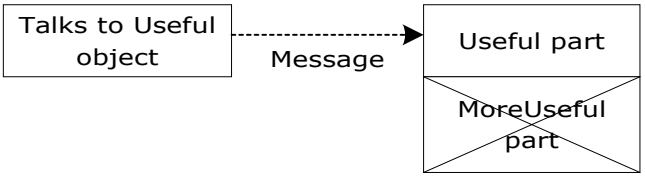


也就是說，base class 可以接收所有發送給 derived class 的訊息，因爲二者具有一模一樣的介面。你只需做一件事，便是將 derived class 向上轉型，而完全不需回頭檢視你所處理的 object 的確切型別。所有事務都可以透過多型（polymorphism）來處理。

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」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 methods 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 methods：

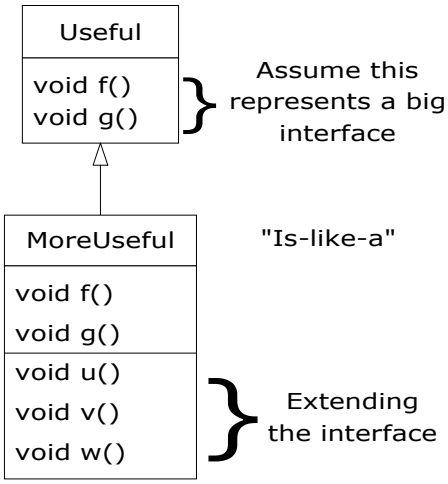


如果你沒有向上轉型，這便不會困擾到你。但你常常會遇到某些情況，使你一定得知道 object 的確切型別，才能夠存取該型別所擴充的 methods。下一節告訴你該怎麼做。

向下轉型（downcasting）與執行期型別判斷（run-tine type identification）

由於向上轉型（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（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」。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 object 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「執行時期型別辨識（*run-time type identification*，RTTI）」。下例說明了 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
  }
} ///:~
```

一如上頁圖中所示，**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 objects 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** object 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 比單純轉型包涵了更多內涵。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 執行時期型別辨識（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 *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、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「method 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 members 與 messages，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，但是值得。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 method 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 method，並觀察所發生的事情。最後，在所有 derived class 中覆寫此一 method。

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，就像作用於舊型別一般。

3. 改寫 **Music3.java**，使 **what()**成爲 root **Object** 的 **toString()** method。請使用 **System.out.println()**（不進行任何轉型）來印出 **Instrument** objects 的內容。

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** 中的方式來隨機產生 **Instrument** objects。

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 methods，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 methods，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 methods 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 member objects，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()** method，而且這些 methods 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 methods，並在第一個 method 中呼叫第二個 method。然後，衍生一個新的 class，並於其中覆寫第二個 method。現在，產生一個 derived class object，將它向上轉型至基礎型別，並呼叫第一個 method。請解釋所發生的事情。

13. 撰寫一個 base class，具有 **abstract print()** method，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 method。現在，在 **main()** 中產生一個 derived class object，然後呼叫其 **print()** method。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 methods。

15. 撰寫一個 **abstract** class，不具任何 methods。自此衍生出一個 class，並爲它加上一個 method。撰寫某個 **static** method，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 method。請在 **main()** 中驗證這是可行的。接著，將此 method 的 **abstract** 飾詞移至 base class，因而免去向下轉型的需要。