《Practical Java》中文版

■敬告讀者

此處開放《Practical Java》中文版最後定稿之部分篇幅（前三章內容，達全書 1/3 篇幅）。此舉得到繁體版出版人碁峰圖書公司與簡體版出版人中國電力出版社之鼎力支持，十分感謝。

本次開放以繁/簡中文讀者爲對象。由於我個人並不直接處理簡體版最終版面工作，因此手上無簡體版之最終電子成品。我所開放的兩份成品，都使用繁體字，惟區分爲「臺灣術語版」和「大陸術語版」。

您目前所見到的這一份成品，是「臺灣術語版」。enjoy it ☺

侯捷 2003.08

Practical Java

Programming Language Guide

中文版

Peter Haggar 著

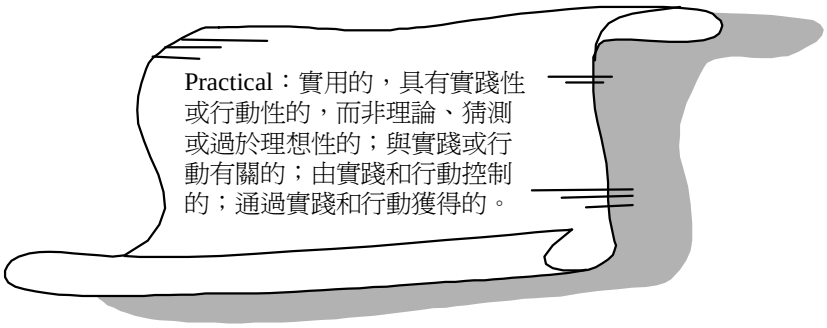
侯捷 / 劉永丹 合譯

教師的影響是永恆的，無遠弗屆且價值難計

— Henry Adams

. . . 紀念我的父親

謹以此書獻給我的妻子 Tara，以及
我的孩子們，Lauren, Keely, 和 Andrew



Practical: Of, relating to,
governed by, or acquired
through practice or action,
rather than theory,
speculation, or ideals

譯序

by 侯捷

面對 Java，可從兩方面看待，一是語言，一是平台。本書談的是 Java 語言，以下我所言種種，也是指 Java 語言。

Java 是一門優秀的物件導向編程語言（Object Oriented Programming Language, OOPL）。什麼是「物件導向」？如何才稱得上「優秀」？前者可定量定性，客觀；後者往往流於個人感受，主觀！所以雖然物件導向語言有著幾近一致的條件和門檻[1]（封裝、繼承、多型…），孰優孰劣卻是各人心中一把尺。儘管如此，無人可以否認 Java 語言在 OOP（物件導向編程）上擁有良好的特性和優越的表現。

我所謂良好的 OOP 特性，指的是 Java 提供了許多讓程式員得以輕鬆表達物件導向技術與思維的語言關鍵字（keywords）如 class, abstract, interface, extends, implements, public, protected, private, final, static, finalize…，又提供條理清晰結構分明的檔案組織方式如 package, import，又擁有嚴謹而靈活的動態型別系統（dynamic type system）使得以提供 RTTI 和 Reflection 機制，並擁有一個優秀、涵蓋面廣、擴充性強的標準程式庫（Java Libraries）。

這些優秀的語言構件（constructs）雖然好用易用，但不論就技術面或應用面或效率考量，還是有許多隱微細節散佈其中，例如 object creation, object initialization, Cloneable, Serializable, Equality, Immutability, Multithreading (Synchronization),

[1] 我常憶起網絡論壇上時可與聞的一種怪誕態度。有一派人士主張，OO 是一種思想，一種思考模式，任何語言都能夠實現它，因而侈言「C 或 assembly 語言也能 OO」。任何語言各有用途，這是完全正確的；OO 是一種思維，這話也是對的。任何語言都能夠實現 OO，這話對某些人也許是對的，對 99.9999%的人是錯的。以 non-OO 語言實現 OO 思維，非但達成度極低，也非人人能爲。Edmund Hillary（艾德蒙 希拉瑞）能達到的高度，你未必達得到 — 事實上你通常達不到。（註：Edmund Hillary 是第一位登上聖母鋒的地球人，1953 年英格蘭遠征隊員。）

Exception Handling…，在在需要 Java 程式員深入認識與理解。

市面上 Java 書籍極多，專注於「編程主題式探討」並「以獨立條款呈現」的書籍比較少。這類書籍面向中高階讀者，不僅選題必須饒富價值、探討必須極爲深刻，各主題最好還獨立以利選擇閱讀，卻又最好彼此前後呼應環環相扣，並附良好交叉索引，予讀者柳暗花明的強烈衝擊。此種「專題條款」式的表現風格，在 Scott Meyers 的《*Effective C++*》和《*More Effective C++*》二書面世之後獲得許多讚揚，也引來許多追隨。

《*Practical Java*》和《*Effective Java*》二書，對前述重要而基礎的技術細微處有著詳盡、深刻、實用的介紹和剖析和範例，又以獨立條款之姿展現，在內容的紮實度、可讀性、易讀性上表現均十分良好。爲此，秉持並承繼我爲 C++ 社群翻譯《*Effective C++*》、《*More Effective C++*》的態度和機緣，我很開心再次由我負責，將《*Practical Java*》和《*Effective Java*》二書中譯本呈獻給 Java 社群。

考慮本書讀者應已具備 Java 編程基礎，對於各種英文術語已有良好的接受度，我在書中保留了許多英文術語，時而中英並陳，包括 class, object, interface, reference, instance, array, vector, stack, heap…，也包括涉及 Java 關鍵字的一些用語如 private, public, protected, static, abstract…，不勝枚舉（下頁另有一個扼要說明）。本書努力在字型變化上突顯不同類形的術語，以利讀者閱讀。本書支援網站有一個「術語‧英中繁簡」對照表，歡迎訪問，網址如下。

《*Practical Java*》由劉永丹先生和我合力完成。永丹做前期初譯工作，我負責後繼的文字修潤、技術檢閱、大局風貌。永丹技術紮實，文字用心。沒有他的協助，本書不可能在這個時間以這樣的品質面世。謝謝永丹。

本書每一章起始處都有作者匠心獨具收集的一些文摘語錄。我們雖勉力譯出，恐見識不足，貽笑大方，故均留下原文和出處，庶幾不誤讀者。

侯捷 2003/07/08 于臺灣.新竹

jjhou@jjhou.com（電子郵箱）
http://www.jjhou.com（繁體）（術語對照表 http://www.jjhou.com/terms.htm）
http://jjhou.csdn.net（簡體） （術語對照表 http:// jjhou.csdn.net/terms.htm）

p.s. 本書已就英文版截至 2003/07/01 之勘誤表修正於紙本。

本書術語翻譯與保留之大致原則：

※ 廣被大眾接受之術語，無需額外說明，不在此列。例如繼承（inheritance）、封裝（encapsulation）、多型（polymorphism）。

※ 本書保留與 Java 關鍵字相關之術語不譯，例如 class, interface, private, public, protected, static, final, abstract, synchronized, serializable…。

※ 本書保留資料結構名稱不譯，例如 array, vector, list, map, set, stack, heap…。"**collection**" 譯爲「群集」。

※ "**class**" 及其所衍生之各種名詞如 subclass, superclass, immutable class, mutable class, base class, derived class 等皆保留不譯（時而英中並陳）。"**object**" 大多數時候譯爲「物件」，時而保留。"**object reference**" 保留不譯，"**reference**" 亦不譯。

※ "**type**" 譯爲「型別」。"**parameter**" 譯爲「參數」，"**argument**" 譯爲「引數」。"**delegate**", "**delegation**" 譯爲「委託」，"**aggregate**", "**aggregation**" 譯爲「聚合」。"**composition**" 譯爲「複合」。

※ 動詞 "**create**" 譯爲「創建」或「建立」，描述物件之初次誕生。動詞 "**refer**" 譯爲「指涉」或「指向」或「引用」。動詞 "**dereference**" 譯爲「提領」。動詞 "**override**" 譯爲「覆寫」。動詞 "**overload**" 譯爲「重載」。

※ 本書將 Java class "**methods**" 譯爲函式，因爲它等價於其他編程語言之 "function"。若直譯爲「方法」，行文缺乏術語突出感，恐影響閱讀流暢；若不譯，過於頻繁出現又恐影響版面觀感。

※ 本書將 Java class "**fields**" 譯爲欄位，等價於 C++ 語言之 "data member"。

※ 本書將 "**clone**" 譯爲「克隆」（這一用詞在中國大陸極爲普遍），映照 "**copy**" 之於「拷貝」。非單純保留 "clone" 是因爲它時常做爲動詞並頻繁出現，而我對術語的保留態度是儘量只考慮名詞（偶有形容詞）。

※「*static* 欄位與 *instance* 欄位」、「*reference* 物件與 *value* 物件」、「*reference* 型別與 *primitive* 型別」等等術語保留部分英文，並使用特殊字型。

※ 本書支援網站有一個「術語‧英中繁簡」對照表，歡迎訪問，網址見上頁。

※ 術語翻譯有許多兩難之處，祈願讀者體諒；譯者勉力求取各方平衡，並儘可能於突兀處中英並陳。

-- 侯捷

譯序

by 劉永丹

拿在您手中的，是一本每一位 Java 程式員都應該關注的書。

要想充分駕馭一門語言（譬如英語），「語法」、「語意」和「語用」缺一不可，即使像 Java 這樣的計算機語言也是如此。Java 稱得上是一門貼心的語言，功能強大卻又不失簡潔優美，「語法」這道門檻似乎輕易就能跨過去。但「語意」往往隱藏在語法之後，需要仔細體察才行，初學者通常需要找來一本優秀教程（教本）仔細研讀和領會。「語用」則涉及多種語言特性的綜合運用，有的深入語言的實現機理（例如 Java 的執行效率），有的甚至超出語言本身的範疇（例如多執行緒）。因此「語用」的掌握絕非一朝一夕之功，無法一蹴而就，最好有位資深專家時刻陪伴左右，耳提面命。

並不是每位程式員都有這樣的福氣，好在有一位資深的 Java 編程專家，Peter Haggar，將其多年的實踐經驗彙集整理，撰寫了您手上的這本《*Practical Java*》。

本書以小品文形式，幫助程式員理解 Java 語言中最核心的「語意」、「語用」專題，因此本書的定位是已初窺門徑的中級程式員。即使您是位 Java 初學者（但一定有隨心所欲駕馭 Java 語言的願望），也請關注這本書，因爲您很快就需要它了。

正如書名所喻示，本書圍繞 Java 編程中遇到的實際問題展開。可以說書中所列專題正是那些令許多 Java 程式員困惑不已的 FAQ（常見問答集）。作者擅長採用恰如其分的示例來闡釋問題，以平實的語言娓娓道出中肯的建議。這些建議往往一語中的，能直接融入編程工作中，可見它們確實是出自作者從實踐中獲得的真知灼見。

本書的另一大特色在於，它並不因爲其實用性而影響討論的深度和說理的透徹，相反，本書對深奧的論題做了聰明的分解，逐步進行細緻入微的探討，這正是作者的高明之處。作者只是以實用性爲界限，摒棄了華而不實、枯燥乏味的理論說教，體現了 Peter Haggar 務實的寫作風格。

與侯捷先生合譯本書是我的榮幸。侯先生對技術嚴謹認真，一絲不苟，爲人卻極其謙和，與其交往的經歷令我受益良多。還要感謝周筠老師，她不斷給與我鼓勵和關注，每一次與她交談都是如沐春風。

劉永丹 上海
2003/05/19

目錄

Contents

實踐：實際履行，尤指藝術、科學或技術領域；與理論遙相對應。

Praxis : Practice, especially of an art, science, or technical occupation; opposite to theory.

— Webster's New Collegiate Dictionary (1958)

細目

Detail Contents

理論上，理論和實際並沒有區別；實際上，它們是有區別的。

In theory, there is no difference between theory and practice. But, in practice, there is.

— Jan L.A. van de Snepscheut

前言
Preface

讓無知儘管信口開河吧，學習自有其價值
Let ignorance talk as it will, learning has its value.
—J. de La Fontaine, *The Use of Knowledge*, Book viii, Fable 19

本書彙集了 Java 編程實踐方面的建議、忠告、範例和討論。本書的組織是一個個獨立課程，每個課程謂之**實踐**（PRAXIS，發音 prak-sis），用以討論特定主題。每個實踐按各自獨立的方式撰寫。你可以從頭閱讀到尾，也可以挑選某些專題閱讀。這種編排風格使你可以在短暫的閒暇中閱讀此書。許多實踐都少於 5 頁，因此你可以在簡短的時間內學習它們。

我在這本書中詳細分析了某些設計（design）和編程（programming）方面的問題。我挑選主題的依據是編程實踐上的有效（effective）和高效（efficient）性質。Java 最被人抱怨的一點是效率（performance），因此我以最大的篇幅討論這一主題，探索使 Java 程式碼執行得更有效率的技術。

我撰寫本書，希望它能夠作爲指南，幫助你設計和撰寫程式。它可以幫助你更全面地理解 Java，讓你撰寫出更高效、更健壯和（或許最重要的）更正確的程式碼。

本書所有資訊都適用於各種 Java 編程，並不囿於伺服端（server）、客戶端（client）或 GUI（圖形用戶界面）編程。此外，你可以將這些資訊運用於 Java 的任一發行版本。

本書風格受到 Scott Meyers 所著的《*Effective C++*》和《*More Effective C++*》的影響[1]。我發現他的風格對書籍組織非常有益，因此我決定採用類似的格式。

預期讀者

本書是爲已經掌握 Java 語言基礎知識的程式員準備的。我假設讀者已經具備 Java 語言和並行編程（concurrent programming）的工作經驗，並理解物件導向（object-oreinted）的概念和術語。本書適用於「想獲得如何高效使用 Java 之實用建議、討論、和範例」的程式員。

無論對 Java 編程老手或新手，本書都爲他們提供了 Java 關鍵領域的資訊和討論。本書提供充足的新資訊，即使經驗豐富的程式員也能從考查他們業已熟悉的領域中獲得極大收益。例如在某些場合，我以獨特的方式討論問題，幫助程式員以不同的方法思考，或使用與以往不同的角度看待事物。

初入門的程式員也可以從本書獲益良多。我提供了討論和範例，幫助他們消除許多常見的編程錯誤。我也澄清了某些常見的 Java 錯誤觀念，並強調語言特性方面的某些問題。

本書組織

本書組織爲六大部分。

1. 一般技術 — 展現 Java 編程的數個基礎領域，包括引數傳遞（argument passing），arrays，`Vectors` 以及垃圾回收（garbage collection）。
2. 物件與相等性 — 研究物件（objects）和基礎型別（primitive types），以及如何、爲何爲一個 class 實作 `equals()`。

[1] *Effective C++: 50 Specific Ways to Improve Your Programs and Designs, Second Edition*, Scott Meyers, Addison-Wesley, 1998. *More Effective C++: 35 New Ways to Improve Your Programs and Designs, Second Edition*, Scott Meyers, Addison-Wesley, 1996.

3. 異常處理 — 提供異常處理技術（exception handling techniques）的細緻分析，並告訴你如何在你的程式碼中高效加入異常處理機制。
4. 效率 — 展示可用來改善程式碼效率的諸多技術，並仔細審查 JVM（Java 虛擬機器）、byte code 和 JITs（Just-in-Time code generators）。
5. 多緒 — 涵蓋執行緒模型（threading model）的諸多方面 — 它們對於建立健壯、可靠的多緒應用程式極爲關鍵。
6. 類別與介面 — 解釋了 interfaces、`abstract` classes 和 concrete classes，以及何處、何時使用它們。本章還詳細討論了 immutable object（不可變物件）、cloning（克隆）和 finalization（終結）。

在上述各標題之下，是數量不等的相關專題。往往我會在不止一處討論特定專題的某個性質。例如我在不同場合討論了 `synchronized` 關鍵字的方方面面，每次討論都涉及 `synchronized` 的不同特性。我提供了數量龐大的交叉參照，你可以由此得知何時閱讀特定專題、何處存在相關資訊。

目錄之後便是細目。這一部分包含所有實踐標題和其頁碼，並附有每個實踐的核心摘要。你可以利用這個細目喚起你對專題的記憶，或用以找出某個專題。

本書附錄內附一份已經證實的技術，可以進一步擴展你的 Java 知識。之後還有一份「進一步閱讀」清單，列出關於 Java、一般設計和編程方面的書籍和期刊。

三言兩語話 PRAXIS（實踐）

`PRAXIS`（實踐）一詞，是我搜尋「得以概括本書所做工作」的詞彙的結果。1982 年的《*American Heritage Dictionary*》將 `PRAXIS` 定義爲：The practical application or exercise of a branch of learning（實際應用或訓練；學習的一個支脈）。這正是我希望在本書中達到的目標。

最確切的恐怕是《*Webster's New Collegiate Dictionary*》於 1958 年給出的一份定義：Practice, especially of an art, science, or technical occupation; opposite to theory（實際履行，尤指藝術、科學或技術領域；與理論遙相對應）。這個定義準確概括了本書精髓。那句 "opposite to theory" 更是畫龍點睛。「理論」本身當然沒錯，但本書沒有爲它準備位置。

範例程式碼

正文所列的所有程式碼，都採用本書寫作時可獲得之 Java 最新版本加以編譯和運行。所有程式碼都曾經在 Windows NT 4.0 環境下以 Sun Java 2 SDK，Standard Edition，v1.2.1 完成編譯和運行。如果你想要得到原始碼，請在以下網址進行註冊：

```
http://www.awl.com/cseng/register
```

該網頁要求你輸入一個獨一無二的碼，此碼可在本書末尾標明爲「How to Register Your Book」的頁面找到。（譯註：中文版讀者可使用這個碼：csng-tbtw-hxqr-xlnz，或自中文版支援網站（見本書封底）之本書專屬網頁下載。）

提供反饋

歡迎讀者對本書提供相關反饋資訊。任何建議、批評或臭蟲報告，都請寄到 PracticalJava@awl.com。

希望本書讓你覺得有用、可讀，並且具實用價值。

Peter Haggar
Research Triangle Park, North Carolina
November, 1999

致謝

Acknowledgment

謝謝你，善良的人，我欠你一份情

Thank you, good sir, I owe you one.

— George Colman, *The Poor Gentleman,* Act I, Scene ii

寫書不是一個人可以獨自完成的。爲了讓本書做成現在這個樣子，許多人付出了他們的時間和精力。審閱書稿可不是件好差事，尤其是那些初期草稿（它們真的是「草稿」）。我真摯地感謝我的審閱人所表現的耐心和熱情。

許多人閱讀、覆閱、（有時）再覆閱（部分）書稿，爲我提供了極具價值和極爲深刻的反饋資訊。這些反饋包括技術乃至語法等各個層面，對於本書內容和脈絡的形成提供了極大幫助。

本書部分草稿由以下人士審閱：Kimberly Bobrow，Joan Boone，Tom Cargill，Bill Field，Dion Gillard，David Hardin，Howard Lee Harkness，Tim Lindholm，George Malek，Jim Mickelson，Devang Patel，Warren Ristow，Susan Elliot Sim，Dan Trieschman。

本書完整書稿由以下人士審閱：Larry Collins，Trevor Cushen，Mary Dageforde，Joshua Engel，Elisabeth Freeman，Cay Horstmann，Susanne Hupfer，Brian Kernighan，Bob Love，Judy Oakley，Linda Rochelle，Clayton Sims，Robert Stanger，Steve Vinoski。

儘管我幸運地擁有如此出類拔萃的審閱人，任何可能的錯誤還是應由我單獨負責。

本書製作過程中，Addison-Wesley 的整個團隊一直是可信賴的引導力量。他們自始至終提供無法估量的幫助和鼓勵，我爲此對他們感激不盡。這個團隊包括我的編輯 Mike Hendrickson，Julie DeBaggis，Tracy Russ，以及 Jacquelyn Doucette。

我也要感謝我在 IBM 的經理，Carolyn Ruby 和 John Feller，感謝他們在我寫作本書的過程中提供的支援。

特別要感謝 Rosemary Simpson，她用嫻熟的技術製作出一流的索引。

我也必須感謝 Wayne Kovsky，Colorado Software Summit Java conference 的創辦人。正是由於我有機會作爲其中的講師，和與會人士有所交流，從中磨礪了許多想法，受到了啓發，才能夠承接這個專案。

同樣要感謝 Larry，David 和 Mark，他們設法補足了我在高爾夫四人對抗賽中的缺席。在我缺席的過去數月中，他們表現出極大耐心。值得讚頌的是，他們保留了我在對抗組中的位置，我急切盼望回到那兒。

就像任何專案一樣，總有一些人特別突出。他們的貢獻遠在他人之上，總是救人於急難。本書寫作期間，他們的詞彙表中似乎沒有 "no" 這個詞。無論何時我尋求幫助、建議、或要求他們就我所重寫的某一段實踐再次覆審，他們總是百忙之中抽出時間提供協助。首先是 Bob Love，他除了提供出色的技術評閱，還提供正確的看法，在我偏離航線時不厭其煩地將我引入正軌。他和 Judy Oakley、Robert Stanger 在實踐 11 至實踐 15 的製作過程中展現出無法估量的價值。Clayton Sims 在本書寫作過程中不但從事審閱，又提供無窮的熱情和鼓勵。我要獻給 Larry Collins 一份特別的感謝，他爲本書的許多方面提供了及時的支援和幫助。他在任何時候都激勵我，並且樂於助人，讓我感激不已。Bob，Clayton 和 Larray 對他們的耳朵也很慷慨大方 — 是的，我不止一次地對他們喋喋不休，嘮叨個沒完。

最重要的，我必須感謝在奮鬥過程中忍受最多的另外四個人：我的妻子 Tara 和三個孩子 Lauren，Keely，Andrew。

最後，我想向我最大的孩子 Lauren 表達謝意，我撰寫本書時她正在學習閱讀。我的「故事」（她這樣稱呼它）終於完成了。

個別實踐（PRAXIS）的致謝

有些人並不知道他們對本書提供了幫助。探究本書內容時，我閱讀了他們的觀點。他們給予我某個實踐的某些思路，下面是這些人士的清單以及他們的工作。

實踐 8 的靈感來自 Sherman Alpert 的文章 "Primitive Types Considered Harmful"，刊載於 *Java Report*, 1998 年 11 月號。

實踐 12 受到 Mark Davis 所寫的 "Java Cookbook: Porting C++ to Java" 一文影響。此文可在以下網址找到：

```
http://www.ibm.com/java/education/portingc/index.html
```

實踐 11 至實踐 15 深深受到我與 Bob Love 和 Robert Sranger 之間爲數眾多、篇幅漫長的討論和電子郵件的影響。

我於 *Java Report*, 1999 年 4 月號上發表文章 "Effective Exception Handling in Java"，之後便收到了來自 Prescott Sanders，Rick Kitts，和 Brian Dellert 的電子郵件。Prescott 和 Rick 提出對 `finally` 區段處理淨化的建議，被我包含於實踐 22。Brian 提出關閉 JIT（turning off the JIT）方面的建議，被我包含於實踐 23。

實踐 27 受到 Tom Cargill 的文章 "Exception Handling: A False Sense of Security" 的啓發，此文發表於 *C++ Report*, 1994 年 11-12 月號。

實踐 33 的靈感來自於 Scott Meyers 的《*Effective C++*》（第 2 版，Addison-Wesley）條款 32。

實踐 35 從 Paul van Keep 的 "Java Performance" 講稿得到啓發，此文於 1998 年在 Keystone, Colorado 舉辦的 Colorado Software Summit Java conference 上發表。

實踐 38 受到文章 "Java: Memories Are Made of This" 的影響，它是 Alex McManus 於 *Java Report*, 1998 年 11 月號發表。

實踐 39 的靈感來自 Kevin Clark 在 1998 ChicagoLand Java Users Group 上所給的講稿。

實踐 43 包含緩式評估（lazy evaluation，延遲求值）的討論和一些範例，它們以 Scott Meyers 的《*More Effective C++*》（Addison-Wesley）條款 17 爲基礎。

實踐 49,51,52 提供的建議，是以在 New York 舉辦的 1999 SIGS Object Expo conference 會議上由 Jonathan Amsterdam 所給的 "Really Understanding Java Threads" 講稿爲基礎。

實踐 54 受到 Alan Holub 於 1998 年 10 月號的 JavaWorld 上發表的 "Programming Java threads in the real world, Part 2" 一文的啓示。此文可於以下網址找到：

```
http://www.javaworld.com/javaworld/jw-10-1998/jw-10-
toolbox.html
```

實踐 56 的靈感來自 "Multithreaded assignment surprises" 一文，那是 Steve Ball 和 John Miller Crawford 在 *Java Report*, 1998 年 9 月號發表的文章。

實踐 59 和實踐 60 受到了 Dion Gillard 的 "Interfaces and Multiple Inheritance in Java" 講稿的影響。此稿發表於 1997 年 Keystone, Colorado 所舉辦之 Colorado Software Summit Java conference。

實踐 67 受到 Bill Field 與我之間數次交談的啓發。

1

一般技術

General Techniques

你可以藉由魅力達到目的，但只能維持 15 分鐘。此後你最好有些真材實料。
You can get by on charm for about 15 minutes. After that, you better know something.
— H. Jackson Brown, Jr., *Live and Learn and Pass It On*

本章涵蓋 Java 編程的數個基本區域。本章詳細闡述 Java 某些方面的問題，這些問題需要額外的解釋和強調，以確保我們可以獲得正確和可理解的程式碼。本章分析比較 `Vectors` 和 arrays 這兩個常用的程式構件，並討論了諸如 `final` 和 `instanceof` 等語言特性，同時包含範例和推薦用法。

本章還討論了 Java 惟一一種引數傳遞機制：*by value*（傳值）。有時候人們會因爲「Java 操控的是 object references」而臆測「引數同樣也以 *by reference*（傳址）方式傳遞」。本章特別指出這種錯誤的想法。本章也探討垃圾回收機制（garbage collection），並提出一種改善記憶體運用的有益技術。

實踐 1：引數以 *by value* 方式而非 *by reference* 方式傳遞

一個普遍存在的誤解是：Java 中的引數以 *by reference* 方式傳遞。這不是真的，引數其實是以 *by value* 方式傳遞。這個誤解源於「所有 Java objects 都是 object references」這一事實（關於 object references 的詳細資訊，請見**實踐 8**）。如果你未能準確理解其中奧妙，會導致一些料想不到的後果。舉個例子：

```
import java.awt.Point;
class PassByValue
{
```

```
    public static void modifyPoint(Point pt, int j)
    {
      pt.setLocation(5,5);                                    //1
      j = 15;
      System.out.println("During modifyPoint " + "pt = " + pt +
                         " and j = " + j);
    }

    public static void main(String args[])
    {
      Point p = new Point(0,0);                               //2
      int i = 10;
      System.out.println("Before modifyPoint " + "p = " + p +
                         " and i = " + i);
      modifyPoint(p, i);                                      //3
      System.out.println("After modifyPoint " + "p = " + p +
                         " and i = " + i);
    }
  }
```

這段程式碼在 //2 處建立了一個 Point 物件並設初值爲(0,0)，並將其值賦予 object reference 變數 p。然後對基本型別 int i 賦予數值 10。//3 呼叫 static modifyPoint()，傳入 p 和 i。modifyPoint()對第一個參數 pt 呼叫了 setLocation()，將其坐標改爲(5,5)。然後將第二個參數 j 賦值爲 15。當 modifyPoint()回返的時候，main()印出 p 和 i 值。這段程式碼的輸出爲何？爲什麼？

程式輸出如下：

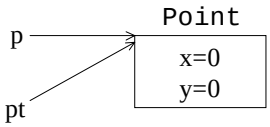
```
Before modifyPoint p = java.awt.Point[x=0,y=0] and i = 10
During modifyPoint pt = java.awt.Point[x=5,y=5] and j = 15
After modifyPoint p = java.awt.Point[x=5,y=5] and i = 10
```

這顯示，modifyPoint()改變了 //2 所建立的 Point 物件，卻沒有改變 int i。在main()之中，i被賦值爲10。由於引數透過 *by value* 方式傳遞，所以modifyPoint()收到 i 的一個副本，然後它將這個副本改爲 15 並回返。main()內的原值 i 並沒有受到影響。

對比之下，你或許認爲 //2 建立的 Point 物件也沒有被 modifyPoint()修改。畢竟 Java 是藉由 *by value* 方式來傳遞引數。於是乎，當呼叫 modifyPoint()並傳入 //2 所建立的 Point 物件時，就會產生一個複件（copy）配合 modifyPoint()

工作。`modifyPoint()`之中對於 `Point` 物件所做的修改不會反映到 `main()`中，因爲它們是兩個不同的物件嘛。對不對？錯！

事實上 `modifyPoint()`是在與「`Point` 物件的 reference 的複件」打交道，而不是與「`Point` 物件的複件」打交道。記住，`p` 是個 object reference，並且 Java 以 *by value* 方式傳遞引數。更明確地說，Java 以 *by value* 方式傳遞 object reference。當 `p` 從 `main()` 被傳入 `modifyPoint()`時，傳遞的是 `p`（也就是一個 reference）的複件。所以 `modifyPoint()`是在與同一個物件打交道，只不過透過別名 `pt` 罷了。在進入 `modifyPoint()`之後和執行 `//1` 之前，這個物件看起來是這樣：



所以 `//1` 執行過後，這個 `Point` 物件已經改變爲`(5,5)`。如果你不同意在諸如 `modifyPoint()`這樣的函式內改變 `Point` 物件，該怎麼辦？有兩種解決辦法：

- 對 `modifyPoint()`傳遞一個 `Point` 物件的克隆件（clone）。關於克隆（cloning）的更具體資訊，請見**實踐 64** 和**實踐 66**。
- 令 `Point` 物件成爲 *immutable*（不可變的）。**實踐 65** 對於 *immutable* 物件技術有所討論。

實踐 2：對不變的 data 和 object references 使用 `final`

許多語言都提供常數資料（constant data）的概念，用來表示那些既不會改變也不能被改變的資料。Java 關鍵字 `final` 用來表示常數資料。例如：

```
class Test
{
  static final int someInt = 10;
  //...
}
```

這段程式碼宣告一個 class static 變數，命名爲 `someInt` 並設其初值爲 10。

任何程式碼如果試圖修改 someInt 將無法通過編譯。例如：

```
//...
someInt = 9;  //錯誤
//...
```

關鍵字 final 可防止 classes 內的個體資料（*instance* data）遭到無意間的修改。如果我們想要一個常數物件，又該如何呢？例如：

```
class Circle
{
  private double rad;

  public Circle(double r)
  {
    rad = r;
  }

  public void setRadius(double r)
  {
    rad = r;
  }

  public double radius()
  {
    return rad;
  }
}

public class FinalTest
{
  private static final Circle wheel = new Circle(5.0);

  public static void main(String args[])
  {
    System.out.println("Radius of wheel is " +
                       wheel.radius());
    wheel.setRadius(7.4);
    System.out.println("Radius of wheel is now " +
                       wheel.radius());
  }
}
```

這段程式碼的輸出是：

```
Radus of wheel is 5.0
Radius of wheel is now 7.4
```

在上述第一個示例中，當我們企圖改變 `final` 資料值，編譯器會偵測出錯誤。在第二個示例中，雖然程式碼改變了 *instance* 變數 `wheel` 的值，編譯器還是爽快地讓它通過了。我們已經明確宣告 `wheel` 爲 `final`，它怎麼還能夠被改變呢？

不，我們確實沒有改變 `wheel` 的值，我們改變的是 `wheel` 所指物件的值。`wheel` 並無變化，仍然指向（代表）同一個物件。變數 `wheel` 是一個 object reference，它指向物件所在的 heap 位置（**實踐 8** 有更多關於 object reference 的資訊）。有鑒於此，下列程式碼會發生什麼事？

```
public class FinalTest
{
  private static final Circle wheel = new Circle(5.0);

  public static void main(String args[])
  {
    System.out.println("Radius of wheel is " +
                       wheel.radius());
    wheel = new Circle(7.4);
    System.out.println("Radius of wheel is now " +
                       wheel.radius());
  }
}
```

編譯這段程式碼，會像我們原本預期地那樣，產生以下錯誤訊息：

```
FinalTest.Java:9 Can't assign a value to a final variable: wheel
    wheel = new Circle(7.4);
    ^
1 error
```

由於我們企圖改變 `final wheel` 的值，因此這個示例產生編譯錯誤。換言之，程式碼企圖令 `wheel` 指向他物。變數 `wheel` 是 `final`，因此也是 *immutable*（不可變的）。它必須永遠指向同一個物件。然而 `wheel` 所指物件並不受關鍵字 `final` 的影響，因此是 *mutable*（可變的）。

關鍵字 `final` 只能防止變數值改變。如果被宣告爲 `final` 的變數是個 object reference，那麼該 reference 不能被改變，必須永遠指向同一個物件。然而被指的那個物件可以隨意改變。*immutable* 物件是**實踐 63** 和**實踐 65** 的主題。

實踐 3：預設情況下所有 non-static 函式都可被覆寫

預設情況下，classes 擁有的任何 non-`private`、non-`static` 函式都允許被 subclasses 覆寫（overridden）。Class 設計者如果希望阻止 subclasses 覆寫（修改）某個函式，必須採取明確動作，也就是將該函式宣告爲 `final`。

關鍵字 `final` 在 Java 中有多重用途，既可被用於 *instance* 變數、*static* 變數（見**實踐 2**），也可用於 classes 或 methods，表示不允許客戶覆寫它們。例如：

```
class Base
{
  public void foo()
  {}
  public final void bar()
  {}
}

class Derived extends Base
{
  public void foo()
  {
    // Overriding Base.foo()
  }
  public void bar()
  {
    // Attempting to override Base.bar()
  }
}
```

編譯上述程式碼，會導致下面的錯誤訊息：

```
Derived.java:15: The method void bar() declared in class Derived
cannot override the final method of the same signature declared in
class Base.  Final methods cannot be overridden.
  public void bar()
              ^
1 error
```

由於 class `Base` 中的 `bar()`被宣告爲 `final`，因此 subclasses 不得覆寫它。這個特性在兩個領域中尤其顯得重要：

- class 設計
- 執行期效率（Runtime Performance）

你或許希望禁止 subclasses 改變某個函式的行爲。例如考慮一個用以表示「螢幕上特定組件」的 class，它具有一個繪製組件用的 `draw()`。由於應用上的嚴格條件，組件外觀不容許改變。你有兩個選擇可以防止 class 用戶修改 `draw()`：

1. 宣告這個 class 爲 `final`。
2. 宣告這個 `draw()`爲 `final`。

宣告某個 class 爲 `final`，也就暗暗宣告了這個 class 的所有函式皆爲 `final`。這種做法可以阻止它衍生 subclasses，從而禁止任何人覆寫此 class 的所有函式。如果這種設計對你過於嚴苛，你也可以考慮只將 `draw()`宣告爲 `final`，這種方式允許衍生 subclasses，並允許後者覆寫該 class 內的任何 non-`final` 函式。無論哪一種情形，`draw()`都不會被某個 subclass 修改，其行爲可確保不變。

設計 classes 時，你可以按此方式將 `final` 當作一種提高效率的工具。宣告某個 class 爲 `final` 之前，請考慮清楚這對 derived classes 帶來的隱含意義和限制。你也必須仔細思考 `final` 函式或 non-`final` 函式對效率的潛在影響。**實踐 36** 探討了相關細節。

實踐 4：在 arrays 和 Vectors 之間慎重選擇

幾乎你所撰寫的任何 Java 程式都很可能運用某種形式的 array。Java 提供兩個看起來相似的構件（constructs）：array 和 `Vector`。事實上兩者全然不同，在你選擇其中某個之前，應該先瞭解它們的某些實際面。

Java arrays 和其他語言的 arrays 幾乎一樣，但它們有些額外優勢。舉個例子，當你建立一個 array 之後，添加的元素個數不能超越 array 大小。在 Java 中，一旦你超出 array 大小，執行期便會產生 `ArrayIndexOutOfBoundsException` 異常。這和其他語言不同，其他語言的執行環境無法標示出這種錯誤，並通常因而導致災難性的後果。

其他語言可運用的「arrays 指標算術運算」（譯註：例如 C++ 允許將 array 名稱加上一個索引整數值），在 Java 中不再成立。Java arrays 是物件，因此你可以透過 array 呼叫 `java.lang.Object` 的所有函式。但如果想知道 array 的容量，不必呼叫某個函式，直接取用其 public 變數 `length` 即可。例如下列程式碼列印出 array 的長度 `ia`：

```
int[] ia = new int[N];
System.out.println("ia length is " + ia.length);
```

arrays 既可以容納基本型別（primitive types）也可以容納 object references（**實踐 8** 告訴你如何區分這兩者）。新建一個 array 時，請記住，每一個元素都將依據其自身型別而被賦予預設值。下表顯示 array 之內不同型別的元素的預設值。

表一：array 之內不同型別的元素的預設值

型別	預設值
`boolean`	`false`
`char`	`'\u0000'`
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0`
`float`	`0.0`
`double`	`0.0`
object reference	`null`

如果 array 的元素是 object references，Java 不會呼叫 *default* 建構式，而是將其初值設爲 `null`。下列程式碼建立了一個 `int` array 和一個 `Button` objects array。程式首先在 array 剛被建立並被賦值之前，列印出所有元素值（預設值），賦值之後再次列印 array 的所有元素值。

```
import java.awt.Button;

class ArrayTest
{
  public static final int arraySize = 3;
  public static void main(String args[])
  {
    int[] ia = new int[arraySize];
    for (int i=0; i<arraySize; i++)
      System.out.println("int array initially " + ia[i]);

    for (int i=0; i<arraySize; i++)
      ia[i]=i+1;

    for (int i=0; i<arraySize; i++)
      System.out.println("int array now " + ia[i]);

    Button[] oa = new Button[arraySize];
    for (int i=0; i<arraySize; i++)
      System.out.println("Button array initially " + oa[i]);

    for (int i=0; i<arraySize; i++)
      oa[i]=new Button("button " + (i+1));

    for (int i=0; i<arraySize; i++)
      System.out.println("Button array now " + oa[i]);
  }
}
```

這段程式碼產生下列輸出：

```
int array initially 0
int array initially 0
int array initially 0
int array now 1
int array now 2
int array now 3
Button array initially null
Button array initially null
Button array initially null
Button array now
java.awt.Button[button0,0,0,0x0,invalid,label=button 1]
Button array now
java.awt.Button[button1,0,0,0x0,invalid,label=button 2]
Button array now
java.awt.Button[button2,0,0,0x0,invalid,label=button 3]
```

對比於 array，當更多元素被加入 `Vector` 以至於超出其容量時，其體積會動態增長，這和 array 有著顯著的不同。此外，`Vector` 於刪除一些元素之後，所有「下

標大於被刪除元素」之所有元素都依次前移，並獲得（比原來小的）新下標。

和 arrays 不同的是，你可以對著一個 Vector 呼叫某函式以得知其大小。Vector 實作了一個 size()，傳回實際元素個數。然而 size()的回傳結果或許和你預期的不同，因爲 size()傳回的是 Vector 內的實際元素數量；若有元素被移除，容器的大小便會改變。array 的大小則是固定不變，無論有多少元素被賦予新值，array 的大小永遠不變。

Vector 的內部其實是以 array 實現的。也就是說當你新建一個 Vector，其內就建立一個 array，並以 java.lang.Object 爲元素型別，用以管理即將被儲存於 Vector 的所有資料項。當 Vector 增長，其內部整個 array 必須重新配置並進行拷貝。當一筆資料（一個元素）從 Vector 中被移除，其底層 array 將被湊實（compacted）。如果未能適當運用 Vector，上述這些和 array 相關的實作性質將會導致效率問題。關於 arrays 和 Vectors 的效率分析，請見**實踐 41**。

最後需要說明的一點是，Vector 只能容納 object references，不能容納基本型別（primitive types）。arrays 則可以容納兩者任一。這個限制是因爲 Vector 運用了「以 java.lang.Object 爲元素型別」的 array 作爲其底部結構。例如下列程式碼：

```
import java.util.Vector;
import java.awt.Button;
class VecArray
{
  public static void main(String args[])
  {
    int i = 1;
    int[] ia = new int[10];
    ia[0] = i;                      //OK

    Button[] ba = new Button[10];
    ba[0] = new Button("");         //OK
    Vector v = new Vector(10);
    v.add(new Button(""));          //OK
    v.add(i);                       //ERROR
    v.add(new Integer(i));          //OK
  }
}
```

會導致這樣的輸出：

```
VecArray.java:15: Incompatible type for method. Can't convert int
to java.lang.Object.
   v.add(i);                 //ERROR
         ^
1 error
```

是的，所有加入 `Vector` 的元素，其型別都必須是 `java.lang.Object` 或其子型別。要讓上述程式碼正常工作，必須將引發錯誤的那一行替換爲：

```
v.add(new Integer(i));
```

如果你需要使用基本型別（primitive types），請考慮以 array 代替 `Vector`。面對基本型別，array 比 `Vector` 擁有更高效率。關於 array、`Vector` 和基本型別的效率特性，詳見**實踐 38**。當你在你的設計和實作中面對 array 和 `Vector` 二擇一時，應當評估二者特性。表二總結了 arrays 和 `Vectors` 的主要不同點。

表二：array 和 `Vector` 的比較

	支援基本型別	支援物件	自動改變大小	速度快
array	Yes	Yes	No	Yes
`Vector`	No	Yes	Yes	No

實踐 5：多型（polymorphism）優於 `instanceof`

Java 提供 `instanceof` 運算子，作爲在執行期確定「某個物件隸屬哪個 class[1]」的機制。就像其他語言特性一樣，`instanceof` 經常被濫用。你可以透過多型（polymorphism）來避免許多常見的 `instanceof` 濫用情況，不僅可以提高程式的物件導向純度，也使程式有較好的擴充性。

[1] 這通常被稱爲物件的執行期型別（runtime type）。譯註：或稱動態型別（dynamic type）

假設你爲公司寫了一個薪資系統。考慮以下的 classes 繼承體系和程式碼：

```
interface Employee
{
  public int salary();
}

class Manager implements Employee
{
  private static final int mgrSal = 40000;
  public int salary()
  {
    return mgrSal;
  }
}

class Programmer implements Employee
{
  private static final int prgSal = 50000;
  private static final int prgBonus = 10000;
  public int salary()
  {
    return prgSal;
  }

  public int bonus()
  {
    return prgBonus;
  }
}

class Payroll
{
  public int calcPayroll(Employee emp)
  {
    int money = emp.salary();
    if (emp instanceof Programmer)
      money += ((Programmer)emp).bonus();  //Calculate the bonus
    return money;
  }

  public static void main(String args[])
  {
    Payroll pr = new Payroll();
    Programmer prg = new Programmer();
    Manager mgr = new Manager();
    System.out.println("Payroll for Programmer is " +
                       pr.calcPayroll(prg));
```

```
      System.out.println("payroll for Manager is " +
                         pr.calcPayroll(mgr));
    }
  }
```

依據這個設計，calcPayroll()必須使用 instanceof 運算子才能計算出正確結果。因爲 calcPayroll()使用了 Employee 介面，所以它必須斷定 Employee 物件究竟實際屬於哪一個 class。程式員有紅利而經理沒有，所以你必須確定 Employee 物件的執行期型別。

這是一個你可以運用多型來代替 instanceof 的鮮明例子。這兒使用 instanceof 毫無道理。萬一你想加入另一個 Employee 型別怎麼辦？假設你需要增加一個 Executive class，在當前設計方案下，你必須改變 calcPayroll()的實作方式，增加一個針對 Executive 的 instanceof 檢查動作。

這樣的設計存在數個問題。首先它缺乏效率，不夠優雅，而且不易擴充。其次它要求程式員撰寫程式碼去做 Java 執行期系統該做的事。是的，程式員（你）呼叫 instanceof 判斷各個物件所屬的 class，以便呼叫適當的函式。更好的方式是令 Java 執行期系統爲你打理這些事務，這可以導出更清晰更優雅的設計，以及更高效、更易於理解的程式碼。

爲了避免上述問題，我們應該按照「不需 instanceof」的方式來組織程式碼。請運用多型以避免 instanceof。此外，有了一個適當設計之後，calcPayroll()就不會因爲新增一個 Employee 型別而必須隨之修改。合適的設計應該使任何時刻都能夠叫用恰當的函式。以下程式碼修正上述問題：

```
interface Employee
{
  public int salary();
  public int bonus();
}

class Manager implements Employee
{
  private static final int mgrSal = 40000;
  private static final int mgrBonus = 0;
  public int salary()
  {
    return mgrSal;
  }
```

```
  public int bonus()
  {
    return mgrBonus;
  }
}

class Programmer implements Employee
{
  private static final int prgSal = 50000;
  private static final int prgBonus = 10000;
  public int salary()
  {
    return prgSal;
  }

  public int bonus()
  {
    return prgBonus;
  }
}

class Payroll
{
  public int calcPayroll(Employee emp)
  {
    // Calculate the bonus. No instanceof check needed.
    return emp.salary() + emp.bonus();
  }

  public static void main(String args[])
  {
    Payroll pr = new Payroll();
    Programmer prg = new Programmer();
    Manager mgr = new Manager();
    System.out.println("Payroll for Programmer is " +
                       pr.calcPayroll(prg));
    System.out.println("Payroll for Manager is " +
                       pr.calcPayroll(mgr));
  }
}
```

在這個設計中，我們爲 `Employee` 介面增加了 `bonus()`，從而消除了 `instanceof` 的必要性。實作出 `Employee` 介面的兩個 classes：`Programmer` 和 `Manager`，都必須實作出 `salary()`和 `bonus()`。這些修改顯著簡化了 `calcPayroll()`。

然而，有時候，這裡所介紹的解決方案行不通，你不得不求助於 `instanceof` 運算子。再次考量最初的 class 繼承體系，假設這次 `Employee` 介面和 `Manager`、`Programmer` 兩個 classes 都是你所使用的程式庫的一部分，而你無法修改其源碼。你的任務是使用這個設計糟糕的程式庫來撰寫 `calcPayroll()`，對不同種類的 `Employee` 計算不同的工資。這種情形下你要不就運用 `instanceof` 運算子撰寫你的 `calcPayroll()`，要不就建立兩個 `calcPayroll()`，一個接受 `Programmer` object references，另一個接受 `Manager` object references。兩種做法都不令人滿意，但如果源碼不可得，我們也只能權宜採用其中一種做法。

`instanceof` 運算子很容易被誤用。很多場合都應該以多型來替代 `instanceof`。無論何時當你看見 `instanceof` 出現，請判斷是否可以改進設計以消除它。以這種方式來改進設計，會產生更合邏輯、更禁得起推敲的設計，以及更容易維護的程式碼。

實踐 6：必要時才使用 `instanceof`

Scott Meyers 在其著作《*Effective C++*》條款 39 中表示，任何時候當你發現自己寫出如此形式的碼：「如果物件的型別是 `T1`，就做某件事，如果物件的型別是 `T2`，就做另一件事」，請賞自己一個巴掌[2]。大多數情形下 Meyers 是正確的，他的賢明建議同樣適用於 Java。然而有時候你還是需要撰寫這樣風格的程式碼。在這些極少數的場合裡，你不必賞自己一個耳光。

實踐 5 曾經提到一種特殊情形：面對一個設計不當的 class 程式庫，用戶可能無法避免使用 `instanceof`。事實上有更多常見情形，使你除了使用 `instanceof` 別無選擇，例如當你必須從一個基礎型別（base type）向下轉型爲衍生型別（derived type）時。幸運的是 Java 提供了安全向下轉型機制，使得不適當的轉型動作導致編譯期錯誤，或在執行期拋出異常。

非法轉型會產生編譯期錯誤，例如從某個型別轉至另一個毫無關係的型別。如果在同一個 class 繼承體系內進行轉型而在執行期被確定爲非法，就會拋出異常

[2] *Effective C++: 50 Specific Ways to Improve Your Programs and Designs, Second Edition*, Scott Meyers, Addison-Wesley, 1998

（exception）。例如：

```
class Shape
{}
class Circle extends Shape
{}
class Triangle extends Shape
{}
class Polynomial
{}
class Shapes
{
  public static void main(String args[])
  {
    Shape shape1 = new Circle();
    Object shape2 = new Triangle();

    Polynomial poly = (Polynomial)shape1;    //Compile error.
    Triangle tri = (Triangle)shape1;         //Runtime error.
    Triangle tri2 = (Triangle)shape2;        //OK
  }
}
```

`shape1` 和 `shape2` 是分別隸屬 `Shape` 型別和 `Object` 型別的兩個 object references。`shape1` 指向一個 `Circle` 物件，`shape2` 指向一個 `Triangle` 物件。

如果你將 reference to `Shape` 轉型爲 reference to `Polynomial`，不能成功，因爲兩個物件之間沒有任何關連。Java 編譯器檢測出這一點，發出編譯錯誤訊息。如果你將 reference to `Shape` 轉型爲 reference to `Triangle`，那麼至少是在同一個物件體系內進行，因此可能合法。但這種向下轉型的合法性只在執行期才得被確認。Java 執行期系統會檢查 `shape1` 所指物件是否爲一個 `Triangle` 物件。由於它不是，遂產生 `ClassCastException` 異常。最後一個轉型動作通過了上述兩項測試，因而合法。

你或許會傾向這樣的想法：一旦需要轉型，只需謹慎行事就可以消除 `instanceof` 的必要性。這不能成立。讓我們考慮 `Vector` class。它具有這樣的性質：可容納以 `java.lang.Object`（或其衍生類別）爲型別的任何元素。此外，從 `Vector` 取得的元素都以 `java.lang.Object` 型別傳回。是的，只要物件儲存於 `Vector` 中，不論該物件屬於哪個衍生型別，都是如此，因爲所有 Java 物件都衍生自 `java.lang.Object`。

爲了運用「儲存於 Vector 內」的 Object 衍生物件所帶函式，你必須將它從 java.lang.Object 向下轉型爲該物件原本隸屬的 derived class。由於 Java 係藉著編譯期和執行期兩道檢查以確保向下轉型的安全性，所以一旦轉型無法正確執行，JVM 就有可能於執行期拋出 ClassCastException 異常。爲避免這種情況，你有兩種選擇：

1. 使用 try/catch 區段，處理 ClassCastException。
2. 使用 instanceof 運算子。

第 1 種做法採用 try/catch 區段，雖然可以運作，但不可取（詳見**實踐 23** 和**實踐 24**）。由於你可以避免發生這種異常，所以更好的解法是採用 instanceof 運算子。這種方法保證獲得「不至於導致執行期異常」的向下轉型。考慮下列程式碼：

```
import java.util.Vector;

class Shape
{}

class Circle extends Shape
{
  public double radius()
  {
    return 5.7;
  }
  //...
}

class Triangle extends Shape
{
  public boolean isRightTriangle()
  {
    //Code to determine if triangle is right
    return true;
  }
  //...
}

class StoreShapes
{
  public static void main(String args[])
  {
    Vector shapeVector = new Vector(10);
    shapeVector.add(new Triangle());
    shapeVector.add(new Triangle());
```

```
      shapeVector.add(new Circle());
      //...
      //Assume many Triangles and Circles are added and removed
      //...
      int size = shapeVector.size();
      for (int i=0; i<size; i++)
      {
        Object o = shapeVector.get(i);
        if (o instanceof Triangle)
        {
          if (((Triangle)o).isRightTriangle())
          {
            //...
          }
        }
        else if (o instanceof Circle)
        {
          double rad = ((Circle)o).radius();
          //...
        }
      }
    }
  }
```

這段程式碼清楚展示了在這種場合下 `instanceof` 運算子是必需的。當程式從 `Vector` 取回物件，它們屬於 `java.lang.Object`。利用 `instanceof` 確定物件實際屬於哪個 class 後，我們便（才）得以正確執行向下轉型，不至於在執行期拋出異常。

另一種選擇是，你可以將 `Circle` 物件和 `Triangle` 物件儲存於不同的 `Vectors` 內，如此便可避免使用 `instanceof`。但如果你無法控制諸如 `Vector` 這樣的群集（collection）的內含物，你就非得使用 `instanceof` 不可。

實踐 7：一旦不再需要 object references，就將它設為 null

Java 垃圾回收機制（garbage collection）負責回收不再被使用的記憶體。程式員因此不再需要明白呼叫某個函式以釋放記憶體。因此，諸如 `free` 和 `delete` 之類的函式都不再必要，Java 也沒有提供它們。垃圾回收機制的存在導致某些程式員忽視記憶體問題。事實上，「記憶體問題純屬庸人自擾」的觀點是一種誤解，記憶體在程式中如何被運用，仍然是值得關注的議題。

一旦你的程式不再參考（或說引用）某個物件，垃圾回收器才會回收這個物件所擁有的記憶體。當你不再需要某物件，你可以將其 references 設爲 `null`，協助垃圾回收器取回記憶體。如果你的程式執行於一個記憶體受限環境中，這可能很有益處。

即使垃圾回收器執行起來，也並非所有「不再被引用」（unreferenced）的記憶體都可被回收。回收效率取決於你的程式碼所處之 JVM（Java 虛擬機器）所採用的垃圾回收演算法。也許需要多次喚起垃圾回收器，才得以收回一個不再被引用的物件。年長且長壽的物件和新生物件相比，似乎比較不容易搖身變爲「不再被引用」，因此常見的垃圾回收演算法總把焦點放在新生物件身上，頻繁地分析它們。

透過「查詢可用記憶體、呼叫 `System.gc()`、再查詢可用記憶體」的步驟，你可以推測你的 JVM 所用的垃圾回收器的某些行爲：

```
Runtime rt = Runtime.getRuntime();
long mem = rt.freeMemory();
System.out.println("Free Memory is: " + mem);
//...
System.gc();
//...
mem = rt.freeMemory();
System.out.println("Free Memory is now: " + mem);
```

當程式呼叫 `System.gc()`，絕大多數 JVM 實作品的回應都是執行垃圾回收器。此外，如果記憶體可用量過低，或 CPU 在一段時間內處於空閒狀態，大多數 JVM 也會隱式（自動地）喚起垃圾回收器。多長久的空閒狀態才會引發垃圾回收器被喚起呢？這取決於 JVM 所採用的垃圾回收演算法。

如果物件內含一些 *instance* 變數並於建構式中爲它們設定初值，而程式用了這樣的物件以致於消耗大量記憶體，就會引發問題。如果這樣的物件存在於程式執行期的大部分時間，而它們所引用的記憶體在這段期間內並不需要，那麼大片記憶體就被浪費掉了。這對你的程式碼效率是有害的。

下面這個例子有個 `main()`，內有一個 `Customers` 物件，負責爲 GUI 提供資訊：

```
class Customers
{
```

```
  private int[] custIdArray;

  public Customers(String db)
  {
    int num = queryDB(db);
    custIdArray = new int[num];
    for (int i=0; i<num; i++)
      custIdArray[i] = //Some value from a database
                       //representing a customer ID
  }
  //...
}

class Foo
{
  public static void main(String args[])
  {
    Customers cust = new Customers("SomeDataBase");
    // use our Customers object to prime fields in a GUI
    // ...
    // Customers object no longer needed.
    // ...
    // The rest of the app...
  }
}
```

假設此處的 Customers 物件很大（也許儲存了 20,000 個顧客識別碼），又假設這個 Customers 物件與程式生命等壽，並且在 GUI 建立完成後不再被需要，而 GUI 的建立距離 main()的結束還有好長一段時間。根據這些，你如何修改程式碼才能不使用這麼多記憶體？

一個解決辦法是在 GUI 建立完成後將區域變數 cust 設爲 null。這麼一來便割斷了對 Customers 物件的引用狀態，垃圾回收器或許便會在下次運行時回收這些記憶體。修改後的 main()看起來像這樣：

```
class Foo
{
  public static void main(String args[])
  {
    Customers cust = new Customers("SomeDataBase");
    // use our Customers object to prime fields in a GUI
    // ...
    // Customers object no longer needed.
    cust = null;
    // The rest of the app...
  }
}
```

如果你需要保持 Customers 物件在整個程式生命期間一直可用，但 GUI 建立後 Customers 物件內的 custIdArray 只是很有限度地被使用，該怎麼做才好呢？未來你或許需要這個 array，但多半情況下不需要，這時候你可以爲 Customers class 提供一個函式，只用來將 custArray 設爲 null，然後在 main()中呼叫該函式：

```
class Customers
{
  private int[] custIdArray;

  public Customers(String db)
  {
    int num = queryDB(db);
    custIdArray = new int[num];
    for (int i=0; i<num; i++)
      custIdArray[i] = //Some value from a database
                       //representing a customer ID
  }

  public void unrefCust()
  {
    custIdArray = null;
  }
  //...
}

class Foo
{
  public static void main(String args[])
  {
    Customers cust = new Customers("SomeDataBase");
    // use our Customers object to prime fields in a GUI
    // ...
    // Customers object no longer needed.
    cust.unrefCust();
    // The rest of the app...
  }
}
```

這段程式碼具有「將 array object reference 設爲 null」的效果。然而這種做法有潛在的負面影響。最初這個 class 的設計是假定存在 array 並具有初值。新增的 unrefCust()改變了這個設計。現在你的程式不得不處理「需要使用這個 array 而它卻不再可用」的局面。或許你還得增加其他新函式，用來在 array 被解除引用（unreferenced）後重新建立它。

這些技術也可以用於 class 函式身上，它們的區域變數可能擁有短暫的生命。別以爲「函式結束時其區域物件不再被引用」所以不會影響你的程式。考慮以下例子：

```
public void someMethod()
{
  BigObject bigObj = new BigObject();
  //Use bigObj
  //Done with bigObj
  bigObj = null;
  //Rest of method...
}
```

將一個區域變數的 reference 設爲 `null` 有一些好處。本例之中 `bigObj` reference 在函式結束前很長一段時間不再被需要，因此，將它設爲 `null` 或許便能在垃圾回收器下次運行時被收回。垃圾回收器的確有可能在這個函式完成之前運行起來。這種情形下如果這個函式的剩餘部分需要大量記憶體，將此區域變數設爲 `null` 應該是有益的。

這些技術可以解決「大量記憶體被佔用而它們其實不再有用」的問題。爲了儘量降低記憶體用量，與程式同壽的物件必須儘可能體積小。此外，大塊頭物件應該儘量「速生速滅」。

限制 Java 程式所佔用的記憶體，或許只對「執行於同一個 JVM 上的 Java 程式」的效率有益，未必有益於相同系統上的其他程式。這是因爲許多 heap 管理演算法會預先配置一些記憶體供你的程式使用。假設你的 JVM 預先配置了 12MB heap，並且不再增長，那麼無論你的程式使用其中多少數量，都不會影響其他系統行程（system processes）的運行。你的程式的 heap 佔用率將直接對自身效率產生影響。

檢閱程式碼時，請注意大塊頭物件，尤其是那些存在於完整（或大部分）程式生命中的物件。你應該仔細研究這些物件的建立和運用，以及它們可能引用多少記憶體。如果它們引用了大量記憶體，請確定是否所有那些記憶體在物件生命期內都真的被需要。也許某些大塊頭物件可以解甲歸田，從而使其後執行的程式碼能夠更高效地運行。

任何時刻你都可以透過 `System.gc()`要求垃圾回收器起身行動。如果你將一個物件解除引用（unreference），你可以呼叫 `System.gc()`要求垃圾回收器立刻運轉，在程式碼繼續執行前先回收被解除引用的那塊記憶體。但是你應當仔細考量這種做法爲你的程式效率帶來的潛在影響。

許多垃圾回收演算法會在它們運轉之前先虛懸（suspend）其他所有執行緒（threads）。這種做法可以確保一旦垃圾回收器開始運轉，能夠擁有 heap 的完整存取權，可以安全完成任務而不受其他執行緒的威脅。一旦垃圾回收器完成了任務，便恢復（resume）此前被虛懸的所有執行緒。

因此，透過 `System.gc()`顯式呼叫，要求垃圾回收器起而運行，你得冒「因執行回收工作而帶來延遲」的風險，延遲程度取決於 JVM 所採用的垃圾回收演算法。

大多數 JVMs 的垃圾回收器都有充足的運行，因此你實在不必明白喚起它。然而如果你的程式碼有些部分期望在繼續進行前先釋放（回收）所有可能的記憶體，你可以考慮呼叫 `System.gc()`。

2

物件與相等性

Objects and Equality

一個孚眾望的「相等性」可以君臨這裡的一切
A popular equality reigns here.
— William Wordsworth, *Complete Poetical Works* (1888)

Java 提供了物件（objects）、基本型別（primitive types）和相等性（equality）三個觀念，這三大要素是進行 class 設計的重要基礎。物件和基本型別的使用方式對程式碼的影響很大，甚至比你想像的還要大。至於相等性則經常被忽視，往往導致 classes 和程式碼夾帶不易察覺的錯誤行爲。噢，用戶當然希望物件的相等性具有穩定可靠的語意。

本章探究這三個要素並提供相關指導，幫助你繞開一些可怕的陷阱，並討論如何實作和使用它們。

實踐 8：區別 reference type 和 primitive types

譯註：本節討論 *reference* 型別和 *primitive* 型別，爲求術語突出和詞性平衡，我在本節保留兩者的英文術語。如果 primitive type 單獨出現，我可能偶而譯爲「基本型別」。

Java 提供了兩種截然不同的型別：*reference*（引用）型別和 *primitive*（基本）型別，後者又稱爲 built-in（內建）型別。每一種 *primitive*（基本）型別分別擁有相應的外覆類別（wrapper classes）。如果你需要一個變數來表示整數，你會選用基本型別 `int` 抑或 `Integer` 物件？如果需要宣告一個布林型別，你會選用基本型別 `boolean` 抑或 `Boolean` 物件？下表列出基本型別和其相應的外覆類別。

表三：基本型別（primitive types）及其相應的外覆類別（wrapper）

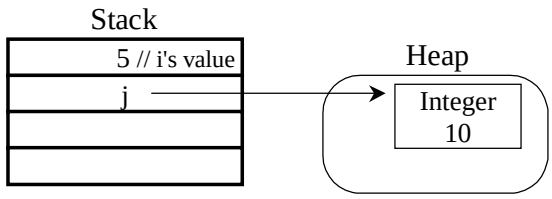
基本型別（Primitive types）	外覆類別（Wrapper class）（首字大寫）
`boolean`	`Boolean`
`char`	`Character`
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

references 和 *primitives* 的行爲方式全然不同，並且具有不同的語意。舉個例子，某函式中有兩個區域變數，其中之一隸屬 *primitives* 型別陣營中的 `int`，另一則是個 *reference*，指向某個 `Interger` 物件：

```
int i = 5;                    // Primitive type
Integer j = new Integer(10);  // Object reference
```

這兩個變數都儲存在區域變數表，它們的操作都在 Java 運算元堆疊（operand stack）中進行，但二者所表述的意義完全不同。（請注意，稍後討論將以術語 stack 表示「運算元堆疊」或「區域變數表」）。不論是基本型別 `int` 或 object reference，都在 stack 中佔據 32 bits 空間[3]，但 `Integer` 物件在 stack 中記錄的並不是物件本身，而是物件的 reference。

所有 Java 物件都藉由 references 來存取，那是某種形式的指標，指向 heap 中的某塊區域，heap 則爲物件的生存提供了真實儲存場所。但如果你宣告一個基本型別，你就是爲它宣告了一份儲存空間。前述兩行程式碼可以這樣表示：



[3] JVM 實作品爲表述一個 `int` 或一個 object reference，最少需要 32 bits 儲存空間。

reference 型別和 *primitive* 型別具有不同的特性和用法。這些不同點涵蓋大小及速度相關問題（詳見**實踐 38**）、型別所儲存的資料結構形式（詳見**實踐 4**），以及「被當作某個 class 的 *instance* 資料」時的預設值。object reference *instance* 變數的預設值是 `null`，primitive type *instance* 變數的預設值則取決於型別 — 請參考 p.8 表一所列 Java 各型別之 *instance* 變數的預設值，那同時也表達了該種型別所構成之 arrays 的預設值。

許多程式員既使用 *primitive* 型別，又使用其外覆類別（wrappers）。測驗相等性（equality）時你必須費一番周折才能同時使用這些型別並對它們如何互相作用和共存有一番認識（參見**實踐 9**）。程式員必須理解這些型別如何運作和相互作用，才能避免寫出漏洞百出的程式。

舉個例子，你不能對一個 *primitive* 型別呼叫函式，但是對一個物件呼叫其所供應的函式，再自然不過了：

```
int j = 5;
j.hashCode();  //ERROR
//...
Integer i = new Integer(5);
i.hashCode();  //OK
```

如果你使用 *primitive* 型別，便免除了「呼叫 new 以創建物件」的需要。這可以節省時間和空間（參見**實踐 38**）。混合使用 *primitive* 型別和物件，也可能在賦值（assignment）時產生意想不到的結果。看起來完全無害的程式碼，所作所爲或許與你預料的並不一致，例如：

```
import java.awt.Point;

class Assign
{
  public static void main(String args[])
  {
    int a = 1;
    int b = 2;
    Point x = new Point(0,0);
    Point y = new Point(1,1);                                    //1
    System.out.println("a is " + a);
    System.out.println("b is " + b);
    System.out.println("x is " + x);
    System.out.println("y is " + y);
    System.out.println("Performing assignment and " +
                       "setLocation...");
```

```
    a = b;
    a++;
    x = y;                                                    //2
    x.setLocation(5,5);                                       //3

    System.out.println("a is " + a);
    System.out.println("b is " + b);
    System.out.println("x is " + x);
    System.out.println("y is " + y);
  }
}
```

這段程式碼產生以下輸出：

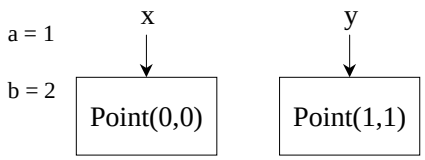
```
a is 1
b is 2
x is java.awt.Point[x=0,y=0]
y is java.awt.Point[x=1,y=1]
Performing assignment and setLocation...
a is 3
b is 2
x is java.awt.Point[x=5,y=5]
y is java.awt.Point[x=5,y=5]
```

對整數 a 和 b 的修改結果沒什麼值得大驚小怪的。變數 a 被賦予 b 的值，然後 a 加 1。輸出結果與我們的預期並無二致。但是在賦值 x 並呼叫 setLocation()之後，x 和 y 的輸出或許令我們感到詫異。經過了 x = y 的賦值動作後，我們又明確地對 x 呼叫了 setLocation()，x 和 y 怎麼還是具有相同的值呢？畢竟我們將 y 值賦給了 x，又改變了 x 值，就像我們對整數 a 和 b 所做的那樣。

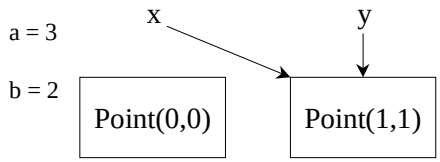
令我們困惑的根源在於「*primitive* 型別」和「物件」的使用方法。賦值動作對二者而言並沒有什麼不同。也許有，但表面上看不出來。賦值使得等號（=）左側值等於右側值。對於諸如上述所提的 int a 和 b 那樣的 *primitive* 型別而言，這是顯而易見的。但對於 non-*primitive* 型別，例如上述的 Point 物件，賦值動作所修改的是 object reference 而不是 object 本身。因此，以下述句之後：

```
x = y;
```

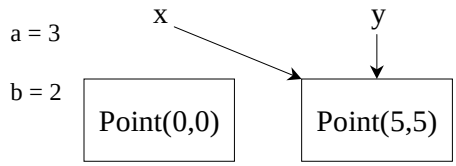
x 等於 y。從另一個角度說，由於 x 和 y 都是 object references，它們現在均指向（代表）同一個物件（object）。於是對 x 的任何改變也同時改變了 y。下面是 //1 程式碼執行後的情形：



經過 `//2` 的賦值動作後，情形如下：



當 `//3` 呼叫 `setLocation()`時，函式作用於 `x` 所指的物件。由於 `x` 和 `y` 指向同一個物件，故而形成：



由於 `x` 和 `y` 指向同一個物件，所有執行於 `x` 身上的函式，就好像執行於 `y` 身上一樣。

弄清楚 *reference* 型別和 *primitive* 型別之間的差異，並理解 references 的語意，至關重要，否則會導致程式碼的行爲和預想的不同。

實踐 9：區分 == 和 equals()

每當討論Java的相等性(equality)，疑惑便如影隨形接踵而來。==運算子和`equals()`函式的區別是什麼？如何取捨？究竟 `equals()`是何方神聖？難道有了== 還不夠嗎？

如果不弄明白這些問題，你的程式碼就有可能臭蟲叢生。考慮下面這個例子：

```
class Test
{
  public static void main(String args[])
  {
    int a = 10;
    int b = 10;
    System.out.println("a==b is " + (a==b));

    Integer ia = new Integer(10);
    Integer ib = new Integer(10);
    System.out.println("ia==ib is " + (ia==ib));
  }
}
```

執行後會產生以下輸出：

```
a==b is true
ia==ib is false
```

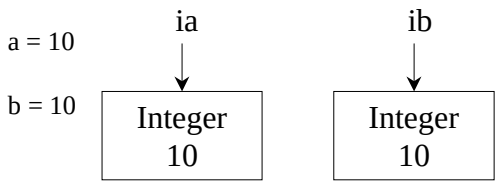
如果你不確知爲什麼產生這樣的結果，把程式碼修改如下或許會清楚一些：

```
class Test
{
  public static void main(String args[])
  {
    int a = 10;
    int b = 10;
    System.out.println("a is " + a);
    System.out.println("b is " + b);
    System.out.println("a==b is " + (a==b));

    Integer ia = new Integer(10);
    Integer ib = new Integer(10);
    System.out.println("ia is " + ia);
    System.out.println("ib is " + ib);
    System.out.println("ia==ib is " + (ia==ib));
  }
}
```

再次執行後，輸出如下：

```
a is 10
b is 10
a==b is true
ia is 10
ib is 10
ia==ib is false
```

如果你還是不清楚爲什麼這段程式碼產生這樣的輸出，讓我們再仔細研究一番。變數 a 和 b 的型別是 int，是一種 *primitive* 型別（參見**實踐 8** 和**實踐 38**），和任何物件都沒有瓜葛。相反地，變數 ia 和 ib 是 object references，指向 Java 的 Integer 物件。你或許會說『喔，可真了不起啊！它們的值相同，都是 10』。唔，你可以說它們相等，也可以說它們不相等。



primitive 型別（基本型別）a 和 b 的確具有數值 10。而作爲 object references 的 ia 和 ib，實際上是指向兩個「其值爲 10 的 Java Integer 物件」。因此，ia 和 ib 的值不是 10；它們的值並不相同，分別代表不同的物件。運算子 == 只是「淺層地」測試相等性。看看運算子左側的東西是不是和右側的東西一樣。由於 ia 和 ib 指向不同物件，這表示它們具有不同的值，所以不相等。

如何測試「ia 和 ib 所指的值」（而非 ia 和 ib 本身）是否相等呢？既然你已經看到 == 運算子不能給予我們想要的結果，這種情況下你應該使用 equals()。它是個函式，所以你只能透過物件使用它，不能對諸如 int、float、boolean 之類的基本型別使用之。欲測試 ia 和 ib 是否 "equals"，程式碼應該這麼寫：

```
class Test
{
  public static void main(String args[])
  {
    Integer ia = new Integer(10);
    Integer ib = new Integer(10);
```

```
    System.out.println("ia.equals(ib) is " + (ia.equals(ib)));
    System.out.println("ib.equals(ia) is " + (ib.equals(ia)));
  }
}
```

這段程式碼產生如下輸出：

```
ia.equals(ib) is true
ib.equals(ia) is true
```

「不同型別之間的比較」是另一個問題來源。考慮下面的程式碼：

```
class Test
{
  public static void main(String args[])
  {
    int a = 10;
    float b = 10.0f;
    System.out.println("a is " + a);
    System.out.println("b is " + b);
    System.out.println("a==b is " + (a==b));

    Integer ia = new Integer(10);
    Float fa = new Float(10.0f);
    System.out.println("ia is " + ia);
    System.out.println("fa is " + fa);
    System.out.println("ia.equals(fa) is " + (ia.equals(fa)));
    System.out.println("fa.equals(ia) is " + (fa.equals(ia)));
  }
}
```

這段程式碼輸出的結果是：

```
a is 10
b is 10.0
a==b is true
ia is 10
fa is 10.0
ia.equals(fa) is false
fa.equals(ia) is false
```

這表示，不同基本型別的數值彼此可能相等，但不同型別的物件則不然。當程式比較基本型別 `int a` 和基本型別 `float b` 時，會將 `int a` 晉升爲一個 `float`，其值從 10 變化爲 10.0，並認爲兩值可被視爲相等。然而，兩個隸屬不同 classes 的物件絕不會被認爲彼此相等，除非你提供一個你自己訂製（編寫）的 `equals()`，但我並不推薦你這麼做。關於 `equals()`實作法的更多資訊，請見**實踐 11** 至**實踐 15**。

這裡的要點是：請使用 == 測試兩個基本型別是否完全相同（*identical*），或測試兩個 object references 是否指向同一個物件；請使用 `equals()`比較兩個物件是否一致（*same*） —基準點是其屬性（attributes。譯註：此處是指物件的實值內容，也就是資料欄位, field）。我們把「根據屬性來比較兩個物件是否相等」稱爲「等值測試」（testing for value），或稱爲「語意上的相等測試」（testing for semantic equality）。在你開始使用 `equals()`之前，請先看看**實踐 10**，那兒揭示了一些不易察覺的問題。

實踐 10：不要依賴 `equals()`的預設實作品

實踐 9 闡釋了何時使用 == 運算子，何時使用 `equals()`。然而如果你對後者的實作方式不聞不問，呼叫它時或許仍然無法獲得你想要的結果。

舉個例子，假設你正在爲某個高爾夫器材批發店撰寫軟體，其中一個任務是計算庫存中的同類球數量。你可能已經爲高爾夫球撰寫了如下的 class：

```
class Golfball
{
  private String brand;        // 品牌
  private String make;         // 型號
  private int compression;     // 彈性

  public Golfball (String str, String mk, int comp)
  {
    brand = str;
    make = mk;
    compression = comp;
  }

  public String brand()
  {
    return brand;
  }

  public String make()
  {
    return make;
  }

  public int compression()
  {
    return compression;
```

```
  }
}
```

每一個 Golfball 物件都具有品牌（brand）、型號（make）和彈性（compression）三個屬性。兩個 Golfball 物件相等意味所有三個屬性都必須相同。進一步假設，每一個 Golfball 物件都被儲存於某資料庫中。日後當你存取資料庫時，必須確定哪些 Golfball 物件是同類型的，才能產生正確的計算。下面是一段用以比較 Golfball 物件的程式碼：

```
class Warehouse
{
  public static void main(String args[])
  {
    Golfball gb1 = new Golfball("BrandX", "Professional", 100);
    Golfball gb2 = new Golfball("BrandX", "Professional", 100);
    //...
    if (gb1.equals(gb2))
      System.out.println("Ball 1 equals Ball 2");
    else
      System.out.println("Ball 1 does not equal Ball 2");
  }
}
```

你已經知道，如果要按照設計好的方式來計算高爾夫球的數量，就必須使用 equals()而不是==（參見**實踐 9**）。然而一旦你執行上一段程式碼，你會驚奇地發現任何 Golfball 物件都不相等。上個例子輸出如下：

```
Ball 1 does not equal Ball 2
```

換句話說這份實作碼的結果就是：任何高爾夫球都不相等。『等一下』，你說，『我剛讀完**實踐 9**，那時你告訴我這一種比較要使用 equals()，現在它卻不管用』。是的，== 只進行 object references 的比較，所以這裡你應該使用 equals()。然而你也知道，每個 Golfball 物件都互相獨立，因此所有的 object references 都不相同。這裡的問題在於：到底你叫用了什麼樣的 equals()？也許你已經猜到，一定不是你想要的那個。

你或許覺得奇怪，**實踐 9** 的程式碼產生了正確的結果，上一段程式碼卻沒有。**實踐 9** 的程式碼對一個 `Integer` 物件呼叫 `equals()`，恰好 `Integer` class 提供了它自己的 `equals()`實作碼，於是就被用上了。本例中的 `Golfball` 物件並沒有實作自己的 `equals()`，因此喚起的是 `java.lang.Object` 的 `equals()`。

請注意，所有 Java classes 都暗自繼承了 `java.lang.Object`，那是 Java 的最基礎類別。`Golfball` class 當然也不例外，而且由於它沒有實作自己的 `equals()`，所以你不知不覺使用了 `java.lang.Object` 提供的 `equals()`。這個函式實作如下：

```
public boolean equals(Object obj)
{
  return (this == obj);
}
```

上頁的 `Warehouse` class 內便是對每一個 `Golfball` 物件執行 `java.lang.Object` 的 `equals()`。這一份 `equals()`預設實作碼並非按照你設想的去做：它只是檢查 object references 是否指向同一個物件[4]。`java.lang.Object` 版本的 `equals()`行爲類似以下動作：

```
if (gb1 == gb2)
//...
```

你知道的，這一條述句在我們的示例中絕不可能爲 `true`。你想知道的其實是高爾夫球的品牌（`brand`）、型號（`make`）和彈性（`compression`）是否相同。欲改善這個問題，`Golfball` class 必須實作出它自己的 `equals()`函式，不能寄望於 `java.lang.Object` 提供的預設實作碼。

```
class Golfball
{
  //As before...
  public boolean equals(Object obj)
  {
```

[4] 你或許奇怪爲什麼 `java.lang.Object` 提供這個函式。答案是，如果不提供，當你對著一個物件呼叫 `equals()`而它未曾實作 `equals()`的話，會導致編譯錯誤，而不是導致錯誤的執行結果。由於其實作僅僅是比較 object references，所以可輕易以這樣的程式碼代替：

```
if (obj1 == obj2)
```

```
    if (this == obj)
      return true;

    if (obj != null && getClass() == obj.getClass())
    {
      Golfball gb = (Golfball)obj;  //Classes are equal, downcast.
      if (brand.equals(gb.brand()) &&  //Compare attributes.
          make.equals(gb.make()) &&
          compression == gb.compression())
      {
        return true;
      }
    }
    return false;
  }
}
```

只要在 `Golfball` class 中按上述方式實作 `equals()`，再執行你的程式碼，就會得到你所期望的正確結果：

```
Ball 1 equals Ball 2
```

請記住，你不一定要檢查物件內的所有欄位（fields），只需檢查必要欄位即可。根據原本的設計，檢驗品牌、型號和彈性就足以確定兩個物件是否相等。如果你認定只要品牌和型號相同就視兩個 `Golfball` 物件爲相等，不計較它們的彈性，那麼你就應當去除 `equals()`函式之中對 `compression` 的檢查。

也請注意 `equals()`實作碼的一些細微處。函式之中對於隸屬基本型別的 `compression` 用的是 `==` 運算子，對 `String` object reference 用的則是 `equals()`。這種做法才能確保獲得正確的結果（參見**實踐 9**）。

到目前爲止你的設計很出色，高爾夫器材批發店的程式運轉得也很好。可是有一天，某位程式員（在閱讀了《*Practical Java*》**實踐 31** 之後☺）打算修改 `Golfball` 物件的 `brand` 和 `make` 欄位，以 `StringBuffer` 取代 `String`。現在 `Golfball` class 變成了這樣子：

```
class Golfball
{
  private StringBuffer brand;
  private StringBuffer make;
  private int compression;
```

```
    public Golfball (StringBuffer str, StringBuffer mk, int comp)
    {
      brand = str;
      make = mk;
      compression = comp;
    }

    public StringBuffer brand()
    {
      return brand;
    }

    public StringBuffer make()
    {
      return make;
    }
    //As before...
  }
```

看起來這是對 class 無傷大雅的修改。你並沒有改變 `equals()`，只是改動 `Golfball` class 的其他部分。`Golfball` class 僅僅從使用 `String` 轉爲使用 `StringBuffer`。現在，完成了修改之後，你會驚訝地看到程式又不能正常運作了。它所產生的輸出如下：

```
Ball 1 does not equal Ball 2
```

這裡發生了什麼事？早先爲了使之正常運轉，你曾經加上 `equals()`。現在它又出問題了，但你對 `equals()`甚至碰都沒有碰一下。你所作的一切就只是將 `String` 變爲 `StringBuffer`，這個問題當然不會是它引起的，是吧？噢，其實就是它造成的。

再看一遍 `equals()`。請注意其中呼叫了另一個 `equals()`。這是個不妙的信號。它呼叫了什麼樣的 `equals()`呢？它呼叫的是 `brand` 和 `make` 的 `equals()`。原例中的這些物件隸屬 `String` 型別，因而喚起的是 `String` 的 `equals()`。現在它們改而隸屬 `StringBuffer` 型別了。問題就在這裡。

Java 的 `String` class 正確實作了一個 `equals()`，但 `StringBuffer` class 根本就沒有實作它。由於這個原因，你一定猜到了，你所喚起的是 `java.lang.Object` 的 `equals()`。這是因爲 `StringBuffer` 的父類別是 `java.lang.Object`，這就說得

通了。我們先前已經討論過，java.lang.Object 的 equals()在此情況下不能用。這也正是你先前自行撰寫 equals()的原因。

這個問題有數個解決方案：

1. 回到使用 String 物件的老路子。
2. 修改 Golfball 的 equals()，使之先將 StringBuffer 物件轉換爲 String 物件，然後再呼叫 equals()。
3. 撰寫一個你自己的 StringBuffer class，其中包含 equals()。
4. 放棄 equals()，撰寫你自己的 compare()函式，用它來比較 StringBuffer 物件的相等性。

選項 1 是最簡單的辦法：回頭使用 String class。你知道它可以勝任，並且只牽涉最少工作。但是出於某些實作方面的考慮（參見**實踐 31**），你或許還是寧願爲 Golfball 物件的 brand 和 make 使用 StringBuffer。

選項 2 仍然使用 StringBuffer，修改 Golfball class 的 equals()，在其中先將 StringBuffer 物件轉換成 String 物件。這種做法保證了在比較動作發生時呼叫的是 String 的 equals()，從而確保結果正確。修改後的 Golfball class 採用如下技術：

```
class Golfball
{
  //As before...
  public boolean equals(Object obj)
  {
    if (this == obj)
      return true;

    if (obj != null && getClass() == obj.getClass())
    {
      Golfball gb = (Golfball)obj;  //Classes are equal, downcast.
      if (brand.toString().equals(gb.brand().toString()) &&
          make.toString().equals(gb.make().toString()) &&
          compression == gb.compression())
      {
        return true;
      }
    }
    return false;
  }
}
```

這個方案確實有效，再一次測試，你的程式的確可以產生正確結果。但是它的實作代價太高了。是的，每次對 StringBuffer 物件呼叫 toString()，都伴隨產生一個新的 String 物件，這代價太高了（見**實踐 32**）。然而它確實解決了由於使用 StringBuffer 而引發的問題。

選項 3 建立你自己的 StringBuffer class，並爲這個 class 實作 equals()。在實作你自己的 equals()之前，請先參見**實踐 11** 至**實踐 15**。這個方法需要多做一些事情，但是你能充分利用 StringBuffer 的優點。

下面是一個名爲 MyStringBuffer 的 class，它利用與 StringBuffer 的 ***has-a***（又名「包含、複合」）關係。這種設計是必需的，因爲 StringBuffer 是一個 final class，而 final classes 不能被擴展（繼承）。因此 MyStringBuffer 物件內含一個 reference 指向一個 StringBuffer 物件。MyStringBuffer 實作如下：

```
class MyStringBuffer
{
  private StringBuffer stringbuf;

  public MyStringBuffer(String str)
  {
    stringbuf = new StringBuffer(str);
  }

  public int length()
  {
    return stringbuf.length();
  }

  public synchronized char charAt(int index)
  {
    return (stringbuf.charAt(index));
  }
  //Create passthrough methods for the rest of the
  //StringBuffer methods as needed.

  public boolean equals(Object obj)
  {
    if (this == obj)
      return true;

    if (obj != null && getClass() == obj.getClass())
    {
      MyStringBuffer sb = (MyStringBuffer)obj; //Classes equal,
```

```
      int len = length();                  //downcast.

      if (len != sb.length())  //If lengths are not equal, strings
        return false;          //can't be either.

      int index = 0;
      while(index != len)       //Compare the strings.
      {
        if (charAt(index) != sb.charAt(index))
          return false;
        else
          index++;
      }
      return true;
    }
    return false;
  }
}
```

現在我們修改 `Golfball` class，令它使用 `MyStringBuffer`：

```
class Golfball
{
  private MyStringBuffer brand;
  private MyStringBuffer make;
  private int compression;

  public Golfball (MyStringBuffer str,
                   MyStringBuffer mk, int comp)
  {
    brand = str;
    make = mk;
    compression = comp;
  }

  public MyStringBuffer brand()
  {
    return brand;
  }

  public MyStringBuffer make()
  {
    return make;
  }

  public int compression()
  {
    return compression;
  }
```

```
  public boolean equals(Object obj)
  {
    if (this == obj)
      return true;

    if (obj != null && getClass() == obj.getClass())
    {
      Golfball gb = (Golfball)obj;  //Classes are equal, downcast.
      if (brand.equals(gb.brand()) &&  //Compare attributes.
          make.equals(gb.make()) &&
          compression == gb.compression())
      {
        return true;
      }
    }
    return false;
  }
  //...
}
```

最後，我們再修改 Warehouse class，令它使用 MyStringBuffer class：

```
class Warehouse
{
  public static void main(String args[])
  {
    Golfball gb1 = new Golfball(
                   new MyStringBuffer("BrandX"),
                   new MyStringBuffer("Professional"), 100);
    Golfball gb2 = new Golfball(
                    new MyStringBuffer("BrandX"),
                   new MyStringBuffer("Professional"), 100);
    //...
    if (gb1.equals(gb2))
      System.out.println("Ball 1 equals Ball 2");
    else
      System.out.println("Ball 1 does not equal Ball2");
  }
}
```

這段程式碼產生的輸出一如我們期待：

```
Ball 1 equals Ball 2
```

選項 4 需要撰寫一個 static compare()函式，用以比較兩個 StringBuffer 物件的相等性。Golfball 的 equals()必須轉呼叫這個 compare()函式，而不再呼叫 equals()。這個 compare()函式的實作類似 MyStringBuffer 的 equals()。

這個方案的好處是：你不必建立另一個 class 來比較兩個 `StringBuffer` 物件。缺點則是你必須修改 `Golfball` 的 `equals()`，使之呼叫 `compare()`函式，代替原本呼叫的 `equals()`。這意味一旦你日後又打算轉而使用 `String` 物件時，`Golfball` 的 `equals()`也必須同時修改。`static compare()`函式和 `Golfball` 的 `equals()` 看起來像這樣：

```
class Golfball
{
  private StringBuffer brand;
  private StringBuffer make;
  private int compression;

  //...
  static boolean compare(StringBuffer sb1, StringBuffer sb2)
  {
    if (sb1 == sb2)
      return true;

    if (sb1 != null && sb2 != null)
    {
      int len = sb1.length();
      if (len != sb2.length())  //If lengths are not equal,
        return false;         //strings can't be either.

      int index = 0;
      while(index != len)
      {
        if (sb1.charAt(index) != sb2.charAt(index))
          return false;
        else
          index++;
      }
      return true;
    }
    return false;
  }

  public boolean equals(Object obj)
  {
    if (this == obj)
      return true;
```

```
    if (obj != null && getClass() == obj.getClass())
    {
      Golfball gb = (Golfball)obj;
      if (compare(brand, gb.brand()) &&
          compare(make, gb.make()) &&
          compression == gb.compression())
      {
        return true;
      }
    }
    return false;
  }
}
```

以上的 equals()故事教給了我們什麼樣的「行事準則」呢？

- 若要比較物件是否相等，其 class 有責任提供一個正確的 equals()。
- 在「想當然耳地呼叫 equals()」之前，應先檢查並確保你所使用的 class 的確實作了 equals()。
- 如果你所使用的 class 並未實作 equals()，請判斷 java.lang.Object 的預設函式碼是否可勝任。
- 如果無法勝任，就應該在某個外覆類別（wrapper class）或 subclass 中撰寫你自己的 equals()。

撰寫自己的 equals()之前，請參考**實踐 11** 至**實踐 15**。

實踐 11：實作 equals()時必須深思熟慮

當你爲某個 class 設計和實作 equals()時，應當考慮幾個問題。首先，你必須仔細考量 class 究竟何時需要提供 equals()。

當「class objects 相等與否」的檢驗工作超過了「object reference 之間的單純比較」時，該 class 就應當提供一個 equals()。從另一個角度說，如果佔用不同記憶體空間的兩個物件有可能被視爲邏輯上相同，那麼這個 class 就應當提供 equals()。

實踐 10 展示了好幾種 Golfball class 的 equals()實作方式。這些 equals()實作法儘管對當時的設計和實作而言是恰當的，並不代表全部做法。

在你撰寫 equals()之前，必須做幾項設計方面的重要決定。你想要讓哪些 classes 的物件與你的 class 物件進行比較？你只打算讓相同 class 的物件之間進行比較嗎？或者你允許 derived class 物件和其 base class 物件比較？接下來你必須決定如何實作 equals()，使它得以提供並厲行上述語意。

你對上述問題的回答，將對 equals()的實作方式和 class 相等性語意產生直接影響。**實踐 12** 至**實踐 15** 討論各種選擇、各種分支後果，以及各種實作方式。

實踐 12：實作 equals()時優先考慮使用 getClass()

強型別引數（strong argument）可以做到「惟有相同 class 所產生的物件才得被視爲相等」。進一步的推論就是：如果兩個物件隸屬不同的 classes 或 types，它們必不相等。對於實作 equals()而言，「惟有相同 class 所產生的物件才得被視爲相等」是一個既清晰又簡明的方案。爲了達成這一點，必須在 equals()實作碼中使用 getClass()。事實上**實踐 10** 的 equals()實作碼中就已經用上了 getClass()。下面是個例子：

```
class Base
{
  public boolean equals(Object obj)
  {
    if (getClass() != obj.getClass())
      return false;
    //...
  }
}
```

getClass()會傳回某個物件的執行期類別（runtime class）。因此，如果上述兩個正在比較的物件並非都隸屬於 Base，equals()便會傳回 false。以下程式碼對於 Base 物件和 Derived 物件會產生程式註解中所說的結果：

```
class Derived extends Base
{}
//...
Base b1 = new Base();
Base b2 = new Base();
Derived d1 = new Derived();
Derived d2 = new Derived();
```

```
if (b1.equals(d1))   //Always false, classes are unequal.
if (d1.equals(b1))   //Always false, classes are unequal.
if (b1.equals(b2))   //Classes are equal, might return true if
                     //attributes are the same.
if (d1.equals(d2))   //Classes are equal, might return true if
                     //attributes are the same.
```

由於 `b1` 和 `d1` 隸屬不同的 classes，以 `getClass()`完成的比較總是回傳 `false`。如果參與比較的物件隸屬同一個 class，那麼一定可以通過 `getClass()`測試，這時的 `equals()`是否傳回 `true`，就取決於這些物件的屬性（attributes，亦即將被拿來比較的欄位, fields）是否相同。因此，一旦使用 `getClass()`就表示不再容許 derived class 物件與 base class 物件被視爲相等 — 也就是說這樣的比較結果總是 `false`。

爲確保 `equals()`正確而直觀的「相等語意」，實作時除了使用 `getClass()`，還需遵循其他準則。針對下面問題，不同的回答會導致不同的 `equals()`實作方式：

1. 「相等」（equality）的內涵是什麼？例如，兩個物件究竟是全部或部分屬性（attributes）一致，才被視爲相等？這個問題稍後馬上就要討論。
2. 如果你正在爲某個 class 實作 `equals()`，這個 class 有 `java.lang.Object` 以外的任何 base classes 嗎？如果有，那些 base classes 曾經實作 `equals()`嗎？這個問題將在**實踐 13** 討論。

第一個問題的回答，可由**實踐 10** 的 `Golfball` 來說明。在那個例子中，爲了比較該種類型的物件是否相等，`Golfball` 提供了一個 `equals()`。如果兩個 `Golfball` 物件相等，它們的三個屬性：`brand`, `make` 和 `compression` 都必須兩兩一致。`Golfball` class 和其 `equals()`採用如下的實作方式：

```
class Golfball
{
  private String brand;
  private String make;
  private int compression;
  public Golfball (String str, String mk, int comp)
  {
    brand = str;
    make = mk;
    compression = comp;
  }
```

```
  public String brand()
  {
    return brand;
  }

  public String make()
  {
    return make;
  }

  public int compression()
  {
    return compression;
  }

  public boolean equals(Object obj)     // 譯註：同 p35 下
  {
    if (this == obj)
      return true;

    if (obj != null && getClass() == obj.getClass())
    {
      Golfball gb = (Golfball)obj;  //Classes are equal, downcast.
      if (brand.equals(gb.brand()) &&  //Compare attributes.
          make.equals(gb.make()) &&
          compression == gb.compression())
      {
        return true;
      }
    }
    return false;
  }
}
```

equals()首先查看參與比較的兩個 object references 是否指向同一個物件，也就是以下程式碼：

```
if (this == obj)
  return true;
```

測試 this 和 obj 是否指向同一個物件。如果是，你可以回傳 true，不必繼續執行餘下的部分。第二個測試：

```
if ((obj != null) && (getClass() == obj.getClass()))
{
  //Compare attributes and return true if they are all the same
}
return false;
```

可以確保你不會對一個 null 物件呼叫 getClass()，並確保參與比較的兩個物件隸屬同一個 class。後一個條件使你可以將此一「型別爲 Object」的物件向下轉型（downcast）爲你所定義的型別，然後再開始對適切的屬性執行相等性檢驗。

Golfball 的 equals()以其設計者對「相等」的定義爲依據，對兩個 Golfball 物件進行精確的相等性檢驗。請注意，這個 class 比較每一個屬性，決定物件是否相等。假如你認爲 compression 不該是這個 class「相等語意」的一部分，那麼，儘管 compression 是 Golfball 的一個有用屬性，你還是可以從 equals()中去除對 compression 的比較。這麼做僅僅只是改變了經由 equals()實作碼而呈現出來的「相等語意」（equality semantics）。

總體而言，equals()的最佳實作方式就是搭配 getClass()，後者可確保只有相同 class 所產生的物件才有機會被視爲相等。此外，equals()應當查看參與比較的物件是否爲同一個物件。equals()不必對每一個屬性（attributes）進行比較，只有那些攸關「相等性」的屬性才需要比較。

實踐 13：呼叫 super.equals()以喚起 base class 的相關行為

爲了讓 equals()產生正確結果，我們經常需要喚起 base class 的 equals()。讓我舉個例子，假設**實踐 12** 中的 Golfball class 是某程式庫的一部分，你正利用它對一個高爾夫器材批發店統計高爾夫球的庫存量。你的程式先從資料庫中讀取資料，然後放置到 Golfball 物件中，再比較這些物件以得到一個計數。這些程式碼運轉良好，但後來你想根據另一個屬性來區分高爾夫球：construction（構造）。由於 Golfball class 是你所使用的程式庫的一部分，你無法直接修改它，因此你建立一個 subclass，爲之增加新屬性，並提供 equals()來比較這個 class 所產生的物件。程式碼看起來像這樣：

```
class MyGolfball extends Golfball
{
  public final static byte TwoPiece = 0;
  public final static byte ThreePiece = 1;
  private byte ballConstruction;
```

```
  public MyGolfball(String str, String mk,
                    int comp, byte construction)
  {
    super(str, mk, comp);
    ballConstruction = construction;
  }

  public byte construction()
  {
    return ballConstruction;
  }

  public boolean equals(Object obj)
  {
    if (this == obj)
      return true;

    if (obj != null && getClass() == obj.getClass())
    {
      MyGolfball gb = (MyGolfball)obj;  //Class equal, downcast.
      if (ballConstruction == gb.construction())
        return true;
    }
    return false;
  }
}
```

你的程式這樣統計 `Golfball` 物件：

```
class Warehouse
{
  public static void main(String args[])
  {
    MyGolfball gb1 = new MyGolfball("BrandX", "Professional",
                                    100, MyGolfball.TwoPiece);
    MyGolfball gb2 = new MyGolfball("BrandX", "Professional",
                                    100, MyGolfball.TwoPiece);
    //...
    if (gb1.equals(gb2))
      System.out.println("Ball 1 equals Ball 2");
    else
      System.out.println("Ball 1 does not equal Ball 2");
  }
}
```

這段程式碼執行起來，會告訴你兩個 `Golfball` 物件相同。這是正確的結果，但如果你令 `Golfball` 物件包含不同屬性，情況如何？像這樣：

```
MyGolfball gb1 = new MyGolfball("BrandX", "Professional",
                                90, MyGolfball.TwoPiece);
MyGolfball gb2 = new MyGolfball("BrandX", "Professional",
                                100, MyGolfball.TwoPiece);
```

在這裡，兩個物件的彈性（compression）不同，但是對它們呼叫 equals()，傳回的結果卻是：

```
Ball 1 equals Ball 2
```

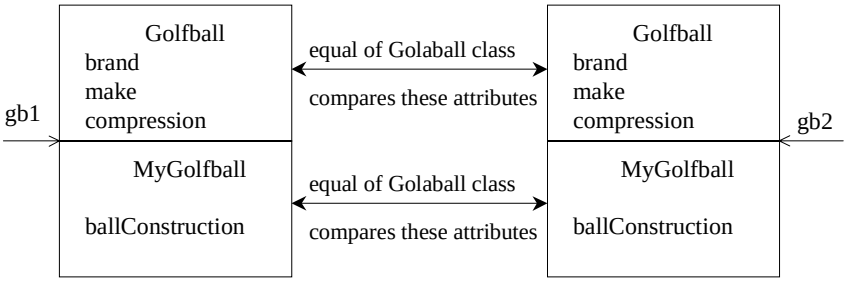
這顯然是不正確的。根據原先設計，這兩個物件並不相等。出了什麼問題呢？

在 Warehouse class 的 main()中，以下程式碼：

```
if (gb1.equals(gb2))
```

喚起 MyGolfball class 的 equals()，這是因爲 gb1 是個 MyGolfball 物件。這個 equals()（只）比較兩個 MyGolfball 物件的 ballConstruction 屬性，由於它們相等，於是回傳 true。base class Golfball 的 equals()並未參與執行。是的，brand,make 和 compression 三個屬性都儲存於 base class 中，沒有參與比較。

要想比較 base class 的屬性，base class 的 equals()必須也被喚起。畢竟，一個 MyGolfball 物件是由 MyGolfball 和 Golfball 兩部分共同構成的，爲求準確無誤地比較這些物件，你必須確保 base class 及其 derived classes 的 equals()都獲得執行。這樣可以確保與「相等性」有關的兩部分（base 成分和 derived 成分）都確實進行了比較。我們可以用下圖表達整個概念：



因此，derived class MyGolfball 的 equals()應當修改，應該在其中呼叫其 base class Golfball 的 equals()：

```
public boolean equals(Object obj)
{
  if (this == obj)                                          //1
    return true;

  if (obj != null && getClass() == obj.getClass() &&        //2
     super.equals(obj))                                     //3
  {
    MyGolfball gb = (MyGolfball)obj;  //Classes equal, downcast.
    if (ballConstruction == gb.construction())  //Compare attrs.
      return true;
  }
  return false;
}
```

MyGolfball 的 equals()在 //3 處做了修改，呼叫 base class Golfball 的 equals()，用以比較 Golfball 的屬性。如果 base class 的屬性（brand,make 和 compression）相等，程式碼接下來才繼續比較 derived class 的屬性 ballConstruction。現在，不論你以相等或不等的物件來執行這段程式碼，都可以產生正確結果。

你或許可以在 derived class 的 equals()實作碼中採用一種簡化方式。上個例子中，//1 和 //2 可以刪掉。由於 super.equals()喚起的是 base class 的 equals()，而後者進行了相同的檢驗，因此兩處相同的檢查顯得多餘。簡化後的 equals()看起來像這樣：

```
public boolean equals(Object obj)
{
  if (super.equals(obj))
  {
    MyGolfball gb = (MyGolfball)obj;  //Classes equal, downcast.
    if (ballConstruction == gb.construction())
      return true;
  }
  return false;
}
```

但是，如果想採用這種簡化方式，你必須首先獲得 base class 的原始碼，確保那兒曾做過適當檢查工作。而且這種簡化方式使你不得不承擔某些風險 — 你的程式有可能因爲未來 classes 繼承體系被修改而受到損壞。例如，一旦「使用這種簡化方式」的某個 derived class 搖身一變爲 base class，其 equals()必須更新，重新添加那些檢驗碼。所以，只有當「刪除那些檢驗碼將會帶來充份的效率提升，而且絕對沒有後續風險」的情況下，才可以使用這種簡化方式。

如果在 Golfball 和 MyGolfball 之間存在另一個 class，會怎麼樣？例如：

```
class Golfball
{
  public boolean equals(Object obj){}
  //As before...
}
class NewGolfball extends Golfball
{
  //Doesn't provide an equals method
}
class MyGolfball extends NewGolfball
{
  public boolean equals(Object obj){}
  //As before...
}
```

由於 MyGolfball 的 equals()呼叫了 super.equals()，所以這段程式碼運作正常並產生正確結果。當你爲一個 derived class 撰寫 equals()時，你必須檢查 java.lang.Object 之外的所有 base classes，看看它們是否都實作有 equals()。如果有，那麼你一定要呼叫 super.equals()。

你得檢查 java.lang.Object 之外的所有 base classes — 這是 java.lang.Object 的 equals()實作方式使然。這個函式僅僅比較兩個 object references 的相等性，而你試圖進行的卻是比較 classes 的某些特定屬性。如果某個 base class 擁有 equals()，就表示它將比較 class 的某些屬性。因此，如果要從 class B 衍生 class D，你一定要在 class D 的 equal()中呼叫 class B 的 equals()，用以比較 B 的屬性。換句話說，一旦 java.lang.Object 之外的 base class 實作有 equals()，你就一定要呼叫 super.equals()。

實踐 14：在 equals()函式中謹慎使用 instanceof

實踐 12 爲我們展示了如何在 equals()函式中使用 getClass()，這種做法使得只有隸屬同一個 class 的物件才能被視爲相等。如果你希望 derived class 物件與其 base class 物件也可被視爲相等，該怎麼做呢？

例如，考慮一個 class 程式庫，它提供了一個代表汽車的 class。這個 class 提供了一些汽車屬性和汽車操縱函式，看起來像這樣：

```
class Car
{
  private String make;
  private int year;
  public void drive()
  {
    //Code to drive the car...
  }
  public boolean equals(Object obj)
  {
    //Compare make and year for equality...
  }
  //...
}
```

假設你產生了一些 Car 物件，在應用程式中使用它們，而且經常需要比較兩個 Car 物件是否相等。你的定義是：兩個 Car 物件的 make（款式）和 year（年份）相同，它們就被視爲相等。後來，你想改變你的某些汽車的駕駛方式。假設你無法修改 Car class 源碼，於是你衍生一個自己的 class，MyCar：

```
class MyCar extends Car
{
  public void drive()
  {
    //Code to drive the car differently than base class, Car
  }
}
```

注意，並沒有新屬性（欄位）加到 MyCar class 中，只不過是增加了一個函式，它覆寫了 drive()。產生 Car 物件時，你可以產生 Car 和 MyCar 兩種物件。記住，你只是修改某些汽車的駕駛方式，因此你應當還是可以比較 MyCar 物件和 Car 物件是否相等。由於參與比較的相關屬性都只存在於 base class Car 之中，不存在於 derived class MyCar 之中，所以你會認爲，原先的相等性比較（equality comparison）還是可以正常運作。

這種情況下，你可以做出結論：「比較 derived class 物件和 base class 物件是否相等」是合情合理的。執行這種比較必須滿足兩個條件：

1. base class 採用 instanceof 來實作 equals()，而不是採用 getClass()。
2. derived class 並沒有實作 equals()。

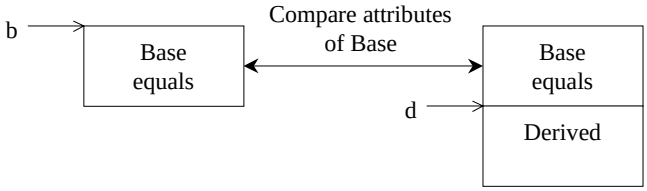
理由如下：

- 如果你希望在 derived class 物件和 base class 物件之間進行相等性比較，base class 的 `equals()`一定不能使用 `getClass()`。因爲如果你使用 `getClass()`，所有從 derived class 產生的物件便都不能視爲與它們的 base class 物件相等。但如果你用的是 `instanceof`，則 derived class 物件可被視爲與其 base class 物件相等（譯註：也就是說，`instanceof` 呈現出「是一種（***is-a***）」語意，而非「絕對是」語意。所以「`MyCar` 是一種 `Car`」這句話成立，但「`MyCar` 就是 `Car`」這句話不成立）。
- 如果 class 沒有實作 `equals()`，那麼你可以假設它沒有新增任何「與相等性有關」的屬性。因此你完全可以倚賴 base class 的 `equals()`函式（它只比較 base class 所含屬性）。
- 如果一個 derived class 實作了 `equals()`，我們可以假設它新增了某些「與相等性有關」的屬性。這些屬性將會被拿來與這個 derived class 的實體（instances）進行比較，或被拿來與「從這個 derived class 再衍生而得的 class 物件」進行比較，但無法與 superclass 物件進行比較。

舉個例子，考慮以下 classes 的定義和宣告：

```
class Base
{
  public boolean equals(Object obj)
  {
    if (obj instanceof Base)
    //...
  }
}

class Derived extends Base
{}

//...

Base b = new Base();
Derived d = new Derived();
if (d.equals(b))  //Potentially true if Base attributes are equal
if (b.equals(d))  //Potentially true if Base attributes are equal
//...
```

這種情形可以下圖表示：



由於 Base 採用 instanceof 實作 equals()而 Derived 沒有實作 equals()，所以，將 Derived 物件與 Base 物件進行比較，就有可能傳回 true，而且這種比較是對稱的（symmetrical）。例如：

```
b.equals(d) == d.equals(b)
```

如果 Base 中的 equals()採用 getClass()進行實作，那便總是傳回 false。針對 b 和 d，getClass()的回傳值並不相同。關於 getClass()詳見**實踐 12**。

如果 base class 和 derived class 都實作了 equals()，情形又當如何？正如前面討論過的，你希望這種比較傳回 false。要知道，derived class 實作出 equals()，表示它具有 base class 並不包含的屬性。考慮下面的例子：

```
class Base
{
  private int attributeOfBase;      //譯註：這個屬性，攸關相等性
  public boolean equals(Object obj)
  {
    if (obj instanceof Base)
      //...
  }
}

class Derived extends Base
{
  private int attributeOfDerived;   //譯註：多出這個屬性，攸關相等性
  public boolean equals(Object obj)
  {
    if (obj instanceof Derived)
    //...
  }
}
```

程式碼 d.equals(b)總是得到 false，原因是 Derived 的 equals()包含了以下程式碼：

```
if (obj instanceof Derived)
```

本例之中 obj 隸屬型別 Base，而非隸屬 Derived，於是導致 false 結果，這正是我們想要的結果。有些程式庫太過馬虎，簡化了「instanceof 檢驗」，在 derived class 的 equals()中僅呼叫 super.equals(obj)，而讓「instanceof 檢驗」在 base class 中完成。簡化後的程式碼類似這樣：

```
class Base
{
  public boolean equals(Object obj)
  {
    if (obj instanceof Base)                              //1
      //...
  }
}

class Derived extends Base
{
  public boolean equals(Object obj)
  {
    if (super.equals(obj))  //Superclass performs instanceof check
    //...
  }
}
```

如果採用這種設計，先前所說的 d.equals(b)就有可能爲 true，並就此失去了「對稱性」（譯註：亦即 d.equals(b)結果爲 true，b.equals(d)結果卻爲 false）。之所以會這樣，是因爲我們將一個 Base 物件傳遞給 derived class 的 equals()，而由於「instanceof 檢驗」已被替換爲一個 super.equals()呼叫動作，於是程式碼喚起 Base 的 equals()，又因爲 Base 物件的確正是 Base 的實體，所以 //1 測試結果爲 true。這樣的程式碼如今竟「容忍」了不合法的比較。一定要記住，如果一個 derived class 實作出 equals()，當其物件與 base class 物件進行比較時，不得回傳 true。只要避免上述所說的簡化形式，就可輕鬆消除這一類問題。

如果 java.lang.Object 以外的 base class 具有 equals()，你還是需要呼叫 super.equals()。這一點已在**實踐 13** 詳細討論過。你應該記住的重點是：仍然寫出「instanceof 檢驗」，並於必要時呼叫 super.equals()。

當我們拿 derived class 物件與 base class 物件進行比較時，存在另一個問題：如果兩個 class 都採用 `instanceof` 實作出 `equals()`，就無法保持比較動作的「對稱性」（譯註：請見上頁倒數第二段的譯註）。就像上頁所說情形，base class 和 derived class 都使用 `instanceof` 實作出 `equals()`。請容我再提醒一次，如果兩個 classes 都擁有 `equals()`，那就意味它們都具有「攸關相等性比較」的屬性。你已經採用「`instanceof` 檢驗」來實作 `equals()`，確保 derived 物件和 base 物件之間的比較總是回傳 `false`，像這樣（譯註：同 p54）：

```
class Base
{
  public boolean equals(Object obj)
  {
    if (obj instanceof Base)
      //...
  }
}

class Derived extends Base
{
  public boolean equals(Object obj)
  {
    if (obj instanceof Derived)
    //...
  }
}

Base b = new Base();
Derived d = new Derived();
```

那麼，執行下一行程式碼：

```
if (d.equals(b))
```

總是會回傳 `false`。原因在於喚起的是 derived class 的 `equals()`，傳遞過去的則是 `Base` 型別。`Base` 物件並非 `Derived` 實體，因此 `instanceof` 檢查結果爲 `false`。

然而，如果你進行以下比較，又將如何：

```
if (b.equals(d))
```

這一次叫用的是 `Base` 的 `equals()`，並傳遞給它一個 `Derived` 型別。由於 `Derived` 物件也**是**（譯註：**是一種**，*is-a*）`Base` 實體，所以 `instanceof` 檢查結果爲 `true`。因此如果 `b` 和 `d` 這兩個物件的 base 成分（屬性）相等，就得到以下結果：

```
if (d.equals(b))  //Always false
if (b.equals(d))  //Always true
```

這種行爲並不直觀，沒有道理。你的 classes 當然不希望這樣。然而對此問題我們並無萬全之策。要想允許 derived class 物件和 base class 物件進行比較，你就不得不承受這個問題的潛在危險。你或許想你可以小心翼翼，並爲大部分 derived class 撰寫測試碼，進而避開這個問題。這種想法帶有以下一些問題：

- 你必須記得撰寫相關測試碼。
- 如果你將 classes 提供給他人使用，他們也必須遵守這些規則。
- 你時常需要通過 base class references 存取物件（譯註：這是多型的表現）。例如從一個 `Vector` 取回一個 derived class 物件，得到的將是一個 `java.lang.Object`，它是個 object reference。如果不執行「會帶來額外開銷」的檢查動作，你就無法知道它到底隸屬哪個 derived class。

於是，實作 `equals()`時你便有了兩種可選方案。你可以在 `equals()`中採用 `getClass()`，在任何情況下保持「對稱的相等性」，像這樣：

```
x.equals(y) == y.equals(x)
```

但需記住，當 derived classes 物件與 base classes 物件進行比較的時候，這種做法總是會得到 `false`。另一個選擇是，你可以在 `equals()`中使用 `instanceof`，允許 derived classes 物件與它們的 base classes 物件（有機會）被視爲相等。

如果你選擇使用 `instanceof`，下面總結三種「在 derived classes 的 `equals()`中採用 `instanceof` 做法」的情形：

1. base class 實作了 `equals()`，而 derived class 沒有。

   假設 base class 的 `equals()`使用 `instanceof`，那麼你可以將 derived class 物件與 base class 物件進行比較。而且這種比較是對稱的（譯註：亦即如果 `b` 相等於 `d`，則 `d` 必然相等於 `b`）。

2. base class 和 derived class 都實作了 `equals()`。

   如果兩個 classes 都在 `equals()`中使用 `instanceof`，當你將 derived class 物件和 base class 物件進行比較時，你希望傳回 `false`。由於 `Base` 物件並非 `Derived` 實體，因此呼叫 derived class `equals()`時的確會傳回 `false`。然而當呼叫 base class `equals()`時，傳回的卻是 `true`。究竟哪一個 `equals()`被喚起，取決於 `equals()`呼叫句中的 object reference 的位置。（譯註：讓我多做一點說明。假設 `b` 是 base class 物件，`d` 是 derived class 物件，`d.equals(b)` 和 `b.equals(d)`分別呼叫的是 derived class 和 base class 的 `equals()`，這是因爲 `b` 和 `d` 位置不同而造成的。）

3. base class 並未實作 `equals()`，但 derived class 實作了它。

   由於 `Base` 物件並非 `Derived` 實體，因此呼叫 derived class `equals()`會傳回 `false`。呼叫 base class `equals()`也會傳回 `false`，但兩者原因並不相同。由於 base class 並未實作 `equals()`，所以喚起的是 `java.lang.Object` 的 `equals()`，它所比較的是兩個 object references 相等與否（參見**實踐 10**）。

真是不幸，「base 物件與 derived 物件之間的比較」竟給 `equals()`的實作方式增添了如此的複雜度，並引入了一些你或許難以接受的問題。

如果快速瀏覽 Java 標準程式庫源碼，你會發現各個 classes 的 `equals()`函式實作碼採用 `instanceof` 的情況很普遍，但也有些採用 `getClass()`。Java 標準程式庫的 classes 並未採取一致方式來實作它們的 `equals()`，此又恰給「相等性比較」的一致化增添了更多難度。

這又引發「在 `equals()`中使用 `instanceof`」的另一個問題。如果你正使用某個程式庫，例如 Java 標準程式庫或某個第三方程式庫（third-party library），你有必要知道你所使用的 classes 如何實作其 `equals()`：它們用的是 `getClass()`還是 `instanceof`？如果用的是 `instanceof`，它們是否在 derived class 中不恰當地呼叫了 `super.equals()`？它們是否視 base class 的實作決定要不要進行「`instanceof` 檢驗」？

如果無法檢驗你所使用的 classes 的源碼，你就必須撰寫測試程式，以便確定那些 `equals()`如何被實作。知道其實作手法，可以對你以及你的客戶發出預警，讓他們知道可能遇上的潛在問題。

此外，如果你正在爲某個 derived class 實作 `equals()`，而其 base class 已經有了一個運用 `instanceof` 完成的 `equals()`，你應當如何實作這個 derived class 的 `equals()`呢？最好的方法是運用 `getClass()`。

藉由運用 `getClass()`，你就可以確信，只有你的各個 derived class 物件才可被視爲相等。你仍然需要呼叫 `super.equals()`來比較 base class 的屬性，但 base class 內存在的「`instanceof` 檢驗」並不會導致這些比較結果呈現 `false`。由於 base class 使用了 `instanceof`，這個 class 仍會呈現「不對稱的相等性」。例如：

```
class Base
{
  public boolean equals(Object obj)
  {
    if (obj instanceof Base)
    //...
  }
}

class Derived extends Base
{
  public boolean equals(Object obj)
  {
    if (obj != null && getClass() == obj.getClass() &&
        super.equals(obj))
    //...
  }
}

Base b = new Base();
Derived d = new Derived();
if (d.equals(b))  //Always false
if (b.equals(d))  //Potentially true
```

真實世界中確實存在一些合情合理的考量，希望程式能夠支援 base class 物件與 derived class 物件之間的相等性比較。然而考慮到使用 `instanceof` 所引發的諸多問題，我還是推薦在 `equals()`實作碼中使用 `getClass()`（參見**實踐 12**）。這樣做就只允許同一個 class 所屬的物件可被視爲相等，進而消除 `instanceof` 做法所帶來的一切後續問題。

實踐 15：實作 equals()時需遵循某些規則

前面探討過的 4 個實踐內容顯示，撰寫一個 equals()並不像你想像的那麼直接而易懂。你得理解各種可能問題，你的 equals()實作碼才能產生正確結果。

無論你選擇使用 getClass()或 instanceof 來實作 equals()，下面的規則適用於所有 equals()函式：

- 如果某個 class 的兩個物件即使佔據不同的記憶體空間，也可被視爲「邏輯上相等」的話，那麼你得爲這個 class 提供一個 equals()。
- 請檢查是否等於 this（參見**實踐 12**）。
- 比較這個 class 中的相關屬性（欄位, fields），以判斷兩個物件是否相等（參見**實踐 12**）。
- 如果有 java.lang.Object 以外的任何 base class 實作了 equals()，那麼就應該呼叫 super.equals()（參見**實踐 13**）。

在 equals()函式中面對 getClass()和 instanceof 進行取捨時，你要仔細斟酌以下問題：

- 如果只允許同一個 class 所產生的物件被視爲相等，通常你應該使用 getClass()（參見**實踐 12**）。
- 只有在不得不「對 derived classes 物件與 base classes 物件進行比較」的場合中，才可以使用 instanceof（參見**實踐 14**），而且你應該明白這樣做帶來的可能問題和複雜性。
- 如果使用 instanceof，而且 derived class 和 base class 都實作有 equals()，你一定要知道，這種比較不會展現出所謂的「對稱相等性」（symmetric equality）。

3

異常處理

Exception Handling

我們已經準備好應付任何可能發生的意外

We are ready for any unforeseen event that may or may not occur.

— Dan Quayle

異常處理（exception handling）是 Java 語言一個強大而實用的特性。但是它的強大也給 Java 增添了一些複雜性。爲了游刃有餘地運用它，我們必須先對這些複雜特性瞭然於胸。Java 爲早先的異常處理模型（exception handling models）引入了新觀念，使異常（exceptions）變得更加容易使用，但同時也使我們更不容易「恰如其份地運用異常」。

異常處理並非萬靈丹 — 儘管我們不時聽到萬靈丹的說法。它不能單槍匹馬解決錯誤處理（error handling）中各方面的疑難問題。除非你知道如何正確使用異常，否則它甚至會引入新問題。「掌握幾個新關鍵字就足以應付錯誤處理」是程式員之間對於異常處理的一種司空見慣的錯誤態度。事實上異常處理僅僅代表錯誤處理的另一條途徑。儘管「異常處理」是傳統「錯誤處理」技術向前發展的重要一步，但如果不能充分理解它，反而會導致在錯誤處理程式碼中增添額外的錯誤。

爲了撰寫容錯（fault-tolerant）、任務極具關鍵性（mission-critical）的軟體，我們需要採用一個有效而強固穩健（robust）的錯誤處理機制和回復（recovery）策略。這不僅有些難度，而且花費時間。如果恰當運用本章內容，你可以令撰寫強固軟體的困難度和花費時間降到最小。如果直到到開發週期的最後階段才來考慮這些事情，你會付出慘重的代價。

首先，**實踐 16** 回顧一些重要的程式流程機制（program flow mechanics），你可以藉此充分理解本章的其他內容。

實踐 16：認識「異常控制流」（exception control flow）機制

欲撰寫能夠正確處理錯誤狀態的強固軟體，你必須深刻理解其中涉及的機制。異常（exceptions）之所以難以應付，一個因素是：它們的行爲類似 `goto` 述句。Java 並沒有提供 `goto` 述句（譯註：其實是保留了這個關鍵字並束之高閣），但異常處理採用了類似技術。一旦某個異常誕生，控制流（control）立刻轉移到下面三者之一：

- `catch` block（捕獲區段）
- `finally` block（終結區段）
- calling method（呼叫端）

這就是異常所表現出的 `goto` 行爲。爲了某些原因，搞清楚這一點很重要。最重要的原因將在**實踐 27** 討論，其中探討了 object state（物件狀態）的關鍵性。另一個原因是你的程式碼可能從某個地點一下子跳轉到另一個地點，這可能導致複雜的邏輯（它們將視程式碼結構而有不同的影響程度），以至於程式碼變得難以除錯（參見**實踐 18**）。考慮以下程式碼：

```
class ExceptionTest {
  public static void main (String args[])
  {
    System.out.println("Entering main()");
    ExceptionTest et = new ExceptionTest();
    try {
      System.out.println("Calling m1()");
      et.m1();
      System.out.println("Returning from call to m1()");     //1
    }
    catch (Exception e) {
      System.out.println("Caught IOException in main()");
    }
    System.out.println("Exiting main()");
  }
  public void m1() throws IOException
  {
    System.out.println("Entering m1()");
    Button b1 = new Button();
    try {
      System.out.println("Calling m2()");
      m2();
      System.out.println("Returning from call to m2()");
      System.out.println("Calling m3()");
```

```
      m3(true);
      System.out.println("Returning from call to m3()");    //2
    }
    catch (IOException e) {
      System.out.println("Caught IOException in " +
                         "m1()...rethrowing");
      throw e;                                              //3
    }
    finally {
      System.out.println("In finally for m1()");
    }
    System.out.println("Exiting m1()");                     //4
  }
  public void m2()
  {
    System.out.println("Entering m2()");
    try {
      Vector v = new Vector(5);
    }
    catch (IllegalArgumentException iae) {
      System.out.println("Caught " +
                         "IllegalArgumentException in m2()"); //5
    }
    finally {
      System.out.println("In finally for m2()");
    }
    System.out.println("Exiting m2()");
  }
  public void m3(boolean genExc) throws IOException
  {
    System.out.println("Entering m3()");
    try {
      Button b3 = new Button();
      if (genExc)
        throw new IOException();
    }
    finally {
      System.out.println("In finally for m3()");
    }
    System.out.println("Exiting m3()");                     //6
  }
}
```

這段程式碼的輸出是：

```
Entering main()
Calling m1()
Entering m1()
```

```
Calling m2()
Entering m2()
In finally for m2()
Exiting m2()
Returning from call to m2()
Calling m3()
Entering m3()
In finally for m3()
Caught IOException in m1()...rethrowing
In finally for m1()
Caught IOException in main()
Exiting main()
```

這裡展現了程式碼的詳細執行軌跡。一旦執行這段程式碼，首先列印一條訊息，然後進入 `try` 區段，再列印一條訊息，然後呼叫 `m1()`。進入 `m1()`之後，列印一條訊息，再進入 `try` 區段列印另一條訊息，然後呼叫 `m2()`。

進入 `m2()`之後，先列印一條訊息，然後進入 `try` 區段，其中並未拋出異常。於是跳過 `catch` 區段，控制流轉移到 `m2()`的 `finally` 區段。在那兒列印一條訊息，並在函式退離（exit）之前列印另一條訊息。`m2()`回返 `m1()`內的呼叫點，又列印出兩條訊息，然後呼叫 `m3()`。

進入 `m3()`之後，列印一條訊息而後進入 `try` 區段，其中拋出一個異常。離開這個函式之前會先執行 `m3()`的 `finally` 區段，列印一條訊息。而後控制流轉移到身爲呼叫端的 `m1()`的 `catch` 區段中。這個 `catch` 區段先列印一條訊息，並再次拋出異常。即將離開 `m1()`之前，先執行 `finally` 區段，列印一條訊息，然後回返至呼叫端 `main()`。

此時進入 `main()`的 `catch` 區段，列印一條訊息。由於異常在那兒獲得了處理，未被再次拋出，所以這個 `catch` 區段結束後，控制流由此繼續向前，並在 `main()`末尾處列印一條訊息。而後程式結束。

這段執行軌跡顯示，一旦程式拋出異常，相應的程式碼立即停止執行，控制流轉移到他處。請注意，本例的 `//1`、`//2`、`//4`、`//5`、`//6` 並沒有出現在輸出之中。

由於程式在 `//1` 和 `//2` 之前產生了異常，控制流直接轉移到函式的 `catch` 區段，因此這兩處的程式碼沒來得及執行。`//4` 和 `//6` 之所以未被執行，原因是此前拋出的異常沒有被捕獲（譯註：`//4` 之前的異常雖然被捕獲了，但再次被拋出，相

當於沒有捕獲），因此在執行完 `finally` 區段後控制流立刻跳離函式（如果`//3` 沒有再次拋出異常，`//4` 會被執行起來）。`//5` 也沒有執行，因爲 `//5` 之前的 `try` 區段並沒有拋出任何異常，也就不可能喚起 `//5` 所在的那個 `catch` 區段。

這說明了當程式在 `try` 區段內拋出異常，將會發生的事情：

- 如果同時存在 `catch` 區段和 `finally` 區段，控制流會先轉移到 `catch` 區段，再跳轉到 `finally` 區段。
- 如果沒有 `catch` 區段，控制流便移轉到 `finally` 區段。

由於 Java 支援垃圾回收機制（garbage collection），你不必爲清理物件記憶體而費心。這也適用於前例中建立的 `Button` 和 `Vector` 物件。如果函式正常結束，或因異常而被迫退離（exit），該函式所創建的所有物件都會自動被解除引用（unreferenced）。因此你不需要手工對這些 object references 解除引用（參見**實踐 7**）。然而你必須明確清理 non-memory 資源（參見**實踐 21** 和**實踐 67**）。

異常處理的另一個概念是：你不能心安理得地對異常視而不見。這也許是好消息，也許是壞消息，取決於看待問題的角度。如果你認爲忽視異常是好主意，或如果你不清楚這樣做的後果如何，請看**實踐 17**。

理解「異常處理的控制流機制」（the mechanics of exception handling control flow），對撰寫強固（robust）的程式而言是絕對必要的。如果你對異常發生時導致的事情懵懵懂懂，會導致程式行爲錯誤，而且難以擴展和維護。

實踐 17：絕對不可輕忽異常（Never ignore an exception）

當異常（exception）誕生時，如果你不捕捉（catch）它，會發生什麼事呢？在 Java 中如果異常產生了卻未被捕獲，發生異常的那個執行緒（thread）將因而中斷。所以你必須對你的程式碼所產生的異常有所作爲。

當 Java 程式產生異常，你能做些什麼呢？

1. 捕捉並處理它，防止它進一步傳播（propagate）。
2. 捕捉並再次拋出它，這麼一來它會被傳播給呼叫端。
3. 捕捉它，然後拋出一個新異常給呼叫端。
4. 不捕捉這個異常，聽任它傳播給呼叫端。

對於 *checked*（可控式）異常，上述選項 2、3、4 要求你將異常添加到函式的 `throw`（拋擲）子句（見**實踐 19** 和**實踐 20**）。選項 1 則遏止異常的進一步傳播。使用選項 3 時，必須確保新拋出的異常包含原異常的相關資訊，這樣才可以保證不丟失重要資訊（參見**實踐 18**）。

常見的情形是，儘管採取了選項 1 的做法，卻以一種草率的方式對異常聽任不管。例如程式碼雖然捕獲異常，卻對它什麼事都不做，像下面這樣：

```
public void m1()
{
  //...
  try {
    //Code that could throw a FileNotFoundException
  }
  catch (FileNotFoundException fnfe)
  {}  //The exception stops here
  //...Rest of code in method
}
```

這段程式碼不理會異常 — 一旦異常發生，只是捕獲它而沒有再次拋出它。請注意，異常沒有得到處理，因此 `catch` 區段之後的程式碼將接下去執行，好像什麼都沒有發生過似的。也就是說，這段程式碼沒有對異常做任何處置，只是將它給「吞」了。吞噬異常比忽略回傳碼（return code）引起更多麻煩，因爲你不得不採用 `try/catch` 區段將「可能拋出異常」的程式碼包裹起來。然而，某種情形下，你的確很可能刻意忽略異常，**實踐 54** 展示了這種情形。

如果你像本例這樣撰寫程式碼，或者和這樣的人共事，請你一定要意識到，這樣做是不明智的，因爲這將沒有留下「異常曾經發生過」的任何記錄。於是你的程式嘗試繼續執行，就像什麼都沒有發生似的。這樣做的危險在於，你的程式可能於稍後失敗，出現更難除錯的情形。如果你在開發初期不知道如何應付異常，至少要像下面這樣做：

```
public void m1()
{
  //...
  try {
    //Code that could throw a FileNotFoundException
  }
  catch (FileNotFoundException fnfe)
  {
    System.out.println(fnfe + " caught in method m1");
    LogException(fnfe);
  }
}
```

這種做法至少提供了一些輸出，以及一份日誌檔（log file），用來記錄程式曾經發生過異常，也讓你知道程式存在一些問題。這種技術也可以用來提醒你，異常已經出現而且暫時沒有適當的回復機制。日後你可以依據日誌檔內的記錄找到相應程式碼，插入合適的異常處理句。注意，你必須提供 `LogException()`，它只需將異常資訊寫入檔案就行了。

這種做法的缺點是：日誌檔僅僅包括測試時發生的異常，無法記錄未發生的異常。所以某次測試過後，即使對日誌檔的所有內容都處理完畢，也並不就意味你已經完全清理好了你的程式。日誌檔僅僅只用來顯示你在測試過程中獲得的異常而已。

如果想要發現測試過程中未曾發生的異常，你必須搜尋整個程式源碼，找出`LogException()`或其他做爲註釋的標籤（tag）。這是確保你不需要使用簡陋的`println()`述句的惟一辦法；那個述句所在之處應該代之以適當的異常處理程式碼。

另一個有效做法是使用 `printStackTrace()`。這個函式提供被拋異常的訊息，以及該異常的起源（以 stack trace 方式呈現），並將它們輸出到標準錯誤串流（standard error stream。譯註：通常是螢幕）。以此做法修改程式碼，獲得的結果是：

```
public void m1()
{
  //...
  try {
    //Code that could throw a FileNotFoundException
  }
```

```
    catch (FileNotFoundException fnfe)
    {
      System.out.println(fnfe + " caught in method m1");
      fnfe.printStackTrace(System.err);
    }
  }
```

使用這項技術時，別忘了將標準錯誤串流輸出設備（standard error stream output）重定向（redirect）至某個檔案。如此一來該檔案就可以用來分析是否發生異常。在 Windows NT 和 UNIX 環境中，下列命令可以將「標準錯誤輸出設備」和「標準輸出設備」重定向至同一個檔案中：

```
java Test > out.dat 2 > &1
```

當然，最好的做法還是不要推諉異常處理和錯誤處理，儘量就地解決它們。

實踐 18：千萬不要遮掩異常（Never hide an exception）

處理先前拋出之異常時，如果 `catch` 區段或 `finally` 區段又拋出異常，某些異常會因此被遮掩，只剩最後生成的那個異常才會傳播給呼叫端。如果你恰好只對「有個函式失敗了，起因是一個或多個異常」這件事感興趣，那麼你或許不在乎是否遮掩先前拋出的異常。但如果你想知道造成函式失敗的「罪魁禍首」，你最好別遮掩異常。

常常，在 `catch` 區段或 `finally` 區段執行過程中，你還會呼叫一些有可能拋出異常的函式。要想不遮掩先前拋出的異常，該怎樣做呢？喔，你需要一種機制，將所有「由某個函式生成的異常」通通傳遞給呼叫端。下面是關於這個問題的一個示例：

```
class Hidden
{
  public static void main (String args[])
  {
    Hidden h = new Hidden();
    try {
      h.foo();
    }
    catch (Exception e) {
      System.out.println("In main, caught exception: " +
                         e.getMessage());
    }
  }
```

```
  public void foo() throws Exception
  {
    try {
      throw new Exception("First Exception");              //1
    }
    catch (Exception e) {
      throw new Exception("Second Exception");             //2
    }
    finally {
      throw new Exception("Third Exception");              //3
    }
  }
}
```

這段程式碼一旦執行，會產生如下結果：

```
In main, caught exception: Third Exception
```

main()之內呼叫了 foo()，然後列印 foo()拋出的異常。輸出結果顯示，這個從 foo()拋出的異常就是 //3 的 finally 區段生成的那個。那麼 //1 的 try 區段和 //2 和 catch 區段產生的異常哪兒去了？

答案是：//1 拋出的異常被 //2 拋出的異常掩蓋了，//2 拋出的異常則被//3 拋出的異常掩蓋了。由於//3 拋出的異常是這個函式以發生時間而言最後拋出的異常，所以它就成爲這個函式最終生成的異常，其他異常都被掩蓋了（丟棄了）。

你或許認爲，這種情形在常規的 Java 程式碼中不會發生。事實上它很普遍。以下例子證明這一點：

```
import java.io.*;
class Hidden
{
  public static void main(String args[])
  {
    Hidden h = new Hidden();
    try {
      h.readFile();
    }
    catch (FileNotFoundException fne) {
      //...
    }
    catch (IOException ioe) {
      //...
    }
  }
```

```
public void readFile() throws FileNotFoundException,
                              IOException
{
  BufferedReader br1 = null;
  BufferedReader br2 = null;
  FileReader fr = null;
  try {
    fr = new FileReader("data1.fil");                    //1
    br1 = new BufferedReader(fr);
    int i = br1.read();                                  //2
    //Other code...
    fr = new FileReader("data2.fil");                    //3
    br2 = new BufferedReader(fr);
    i = br2.read();                                      //4
    //Other code...
  }
  finally {
    if (br1 != null)
      br1.close();                                       //5
    if (br2 != null)
      br2.close();                                       //6
  }
 }
}
```

這例子有個 `readFile()`，用來從檔案讀取資料。它可能拋出兩種異常：一種是 `FileNotFoundException`，可能在 `//1` 和`//3` 發出，另一種是 `IOException`，可能在 `//2`、`//4`、`//5`、`//6` 產生。這個函式也擁有一個 `finally` 區段，負責在函式退離前一刻關閉 `BufferedReader` 物件。這些呼叫動作安置於 `finally` 區段中，可以確保它們一定會被執行（參見**實踐 16** 和**實踐 21**）。然而 `finally` 區段中的 `close()`也可能產生 `IOException`，這就是問題根源。

這段程式碼在某種情形下會掩蓋異常。當`//3` 產生了一個 `FileNotFoundException` 時，控制流（control flow）轉移到 `finally` 區段，其中負責關閉 `br1`。如果關閉 `br1` 的動作導致 `IOException`，會發生什麼事？果真如此，傳回給呼叫端的將會是 `IOException` 而不是 `FileNotFoundException`。呼叫端將無法知道它所呼叫的函式的最初失敗原因是「未能找到某個檔案」（`FileNotFoundException`）。

如果發生異常，控制流（control flow）轉移到某個 `catch` 區段或 `finally` 區段，而那裡又產生新異常，我們該怎麼辦？記住，任何地方都有可能產生異常，而且這些異常在 `catch` 區段和 `finally` 區段中仍舊有活性。此外 `try/catch/finally` 區段之間可以相互任意嵌套（nest）。

對此問題的一個解決方法是，保存一個 list，用以包含所有異常。然後拋出一個異常，其中持有一個 reference 指向上述 list。採用這個辦法，新異常的接受者就擁有了異常的全部資訊，並且不會丟失關鍵錯誤資訊。

先前的程式碼修改如下，包含一個新 class 用來容納 `readFile()`產生的所有異常。`readFile()`將它產生的所有異常加入一個 `Vector` 內，並在 `ReadFileExceptions` 物件中保存一個 reference 指向該 `Vector`。修改後的程式碼是：

```
import java.io.*;
import java.util.Vector;

class ReadFileExceptions extends IOException
{
  private Vector excVector;
  public ReadFileExceptions(Vector v)
  {
    excVector = v;
  }
  public Vector exceptionVector()
  {
    return excVector;
  }
  //...
}

class NotHidden
{
  public static void main(String args[])
  {
    NotHidden nh = new NotHidden();
    try {
      nh.readFile();
    }
    catch (ReadFileExceptions rfe) {
      //...
    }
  }

  public void readFile() throws ReadFileExceptions
  {
    BufferedReader br1 = null;
    BufferedReader br2 = null;
    FileReader fr = null;
    Vector excVec = new Vector(2);  //Vector to store exceptions
```

```
      try {
        fr = new FileReader("data1.fil");
        br1 = new BufferedReader(fr);
        int i = br1.read();
        //Other code...
        fr = new FileReader("data2.fil");
        br2 = new BufferedReader(fr);
        i = br2.read();
        //Other code...
      }
      catch (FileNotFoundException fnfe) {
        excVec.add(fnfe);  //Add exception to Vector
      }
      catch (IOException ioe){
        excVec.add(ioe);  //Add exception to Vector
      }
      finally {
        if (br1 != null)
        {
          try {
            br1.close();
          }
          catch (IOException e) {
            excVec.add(e);  //Add exception to Vector
          }
        }
        if (br2 != null)
        {
          try {
            br2.close();
          }
          catch (IOException e) {
            excVec.add(e);  //Add exception to Vector
          }
        }
        if (excVec.size() != 0)
          throw new ReadFileExceptions(excVec);//Pass all exceptions
      }                                        //to caller.
    }
  }
```

在這段程式碼中，`readFile()`建立了一個 `Vector`，用以保存異常。函式所產生的任何異常都會被加入這個 `Vector` 中。函式退離之前會檢查這個 `Vector` 是否有什麼東西被加進來了。如果有，就建立一個 `ReadFileException` 物件，並將那個「用以保存函式所生異常」的 `Vector` 一併傳遞給它。另一種做法是，你可以只在發生異常的時候才建立 `Vector`，這可以避免由於「建立 `Vector` 卻未使用」而造成一些無謂開銷。「建立物件」所帶來的開銷和「未用上的物件」（unused objects）所花費的代價，在**實踐 32** 和**實踐 33** 中分別有詳細討論。

如果對這個問題考慮不周，會帶來一些負面後果，主要是丟失原始異常（original exception），而且處理「最終被拋出之異常」的程式碼對此前拋出的所有異常毫不知情。如果程式碼試圖根據異常進行回復，卻對發生過的其他異常一無所知，是極爲不利的。

這段程式碼顯示，`try`、`catch` 和 `finally` 區段可以彼此任意嵌套（nested）。此外，儘管從它們之中拋出了多個異常，但只有一個異常可被傳播到外界。記住，最後被拋出的異常是惟一被呼叫端接收到的異常，其他異常都被掩蓋而後遺失了。如果呼叫端需要知道造成失敗的初始原因，程式之中就絕不能掩蓋任何異常。

實踐 19：明察 `throws` 子句的缺點

`throws`（拋擲）子句是一種語言機制，用來列出「可從某個函式傳至外界」的所有可能異常。編譯器強迫你必須在函式中捕捉這些被列出的異常，否則就得在該函式的 `throw` 子句中宣告它們。`throws` 子句用來向函式呼叫者發出預警，告知將會產生哪些異常。這是一項非常有用的語言特性，提供函式用戶非常有價值的資訊（參見**實踐 20**）。然而這個特性也有副作用，你必須心裡有數。

當你對一個大專案的撰碼成果感到滿意的時候，如果又給一個低階的所謂「工蜂型函式」（譯註：原文爲 worker method，取其義）增加一個可能拋出的異常，情況會怎樣？爲方便討論，這裡所謂「工蜂型函式」（worker method）指的是那些爲其他眾多函式服務的函式，它們完成共通性任務，在程式碼的許多地方被呼叫。典型的系統擁有許多這樣的函式。如果上述情形發生，你面臨兩種選擇：

1. 在「工蜂型函式」中捕獲異常，並且就地處理。
2. 從「工蜂型函式」拋出異常，讓呼叫者處理它。

由於「工蜂型函式」或許沒有能力獨立處理這個異常，上述第一選項可能行不通（當然啦，一切取決於系統的設計）。除了採用第二選項，拋出這個異常，你別無選擇。但這或許不像看起來那麼容易。

將一個異常加入函式的 throws 子句，會影響該函式的每一個呼叫者 — 所有呼叫這個「工蜂型函式」的函式統統需要修改。他們將面臨相同的兩個選擇，一如上頁所列。如果它們決定處理異常，異常就不會進一步傳播。然而如果其中某些函式無法處理異常，該怎麼辦？它們必須指望第二條出路，也就是在它們的 throws 子句中增加這個異常。做完這些修改後，你重新編譯你的程式碼。當編譯器發出錯誤訊息時，你或許明白你又一次面臨窘境。現在，所有「呼叫了你剛剛修改過的那個函式」的函式(s)，又得在相同的兩個選項（一如上頁所列）之間做出抉擇。如果一直沒有某個函式挺身而出處理該異常，這個過程會一路延續，直到上溯至 main()才終了。

如果你希望儘量降低遇到這個問題的機率，就請不要在開發週期的最後才添加異常的處理。請在一開始就設計好你的錯誤處理策略。如果你在精心計劃後仍然遇到了這種情形，你也終於完全明白了它的後果。throws 子句是一種實用而有益的語言特性，但如果你不夠細心，它會令你頭痛不已。

實踐 20：細緻而全面地理解 throws 子句

提供 throws 子句的用意在於，提醒函式呼叫者，告知可能發生的異常。明白了這一點之後，讓我們考慮以下程式碼，其中宣告三種異常，以及一個「可能拋出所有這三種異常」的函式：

```
class Exception1 extends Exception  {}
class Exception2 extends Exception1 {}
class Exception3 extends Exception2 {}
class Lazy
{
  public void foo(int i) throws Exception1
  {
    if (i==1)
      throw new Exception1();
    if (i==2)
      throw new Exception2();
    if (i==3)
      throw new Exception3();
  }
}
```

foo()宣告了一個 throws 子句，其中只列出一種異常型別。可是這個函式卻拋出了三個不同的異常。這段程式碼能順利通過編譯嗎？我們可以這麼寫嗎？

這段程式碼完全合法，可順利通過編譯，不會發出任何警告。不過，以這種方式撰碼帶有一些缺點。foo()拋出的三個異常其實都隸屬於型別 Exception1。Exception2 和 Exception3 乃衍生自 Exception1。由於符合 throws 子句的要求，才得以順利通過編譯。這個 throws 子句向呼叫者發出預警說：函式可能產生 Exception1 型別的異常。沿續此例，爲 foo()撰寫 throws 子句的更好寫法是像下面這樣：

```
public void foo(int i) throws Exception1, Exception2, Exception3
```

這個 throws 子句明確列出了函式可能產生的所有異常。回頭看看原先的 foo()：

```
public void foo(int i) throws Exception1
```

上述 throws 子句通知 foo()呼叫者說：函式可能產生的異常乃是隸屬於 Exception1 型別。這當然不能算錯，但事實上這個函式還產生了 Exception2 和 Exception3 兩種型別的異常。這個函式的呼叫者僅僅知道有 Exception1 這一回事。呼叫者如果想進一步瞭解這個函式所產生的衍生異常（derived exceptions），惟一辦法是查看源碼。這是不切實際的，通常無法辦到。於是，呼叫 foo()的程式碼往往看起來是這樣：

```
//...
try {
  Lazy l = new Lazy();
  l.foo(1);
  //...
}
catch (Exception1 exc)
{
  //Handle it...
}
```

當函式拋出異常，導致上述的 catch 區段起而執行的時候，你或許需要處理 Exception1、Exception2 或 Exception3 異常。如果你能取得源碼，你可以透過 instanceof 檢查被拋異常的真正型別。然而，使用 instanceof 既笨拙又累贅而且麻煩。而且就算你能夠拿到源碼，你知道如何檢查嗎？如果你得不到源碼，你就只能處理「距離最近的」衍生異常（least derived exception class），並祈禱它封裝了「實際的」衍生異常的相關資訊。（譯註：所謂「距離最近的」衍生異常，係指最先衍生出來的類別，例如上例的 Exception1 就是可通過 throws 子句的最上層型別。）

當你給出 throws 子句的時候，要填寫得完整無缺。雖然編譯器不強求這樣，但將函式可能拋出的所有異常統統列出，是良好的編程習慣。不要對「這些異常都衍生自另一個異常」的事實戀戀不捨，過於罣礙，因爲出現這種情形往往意味它們代表了大相逕庭的錯誤條件。throws 子句即使再囉嗦一點兒，也只不過是要你多打幾個字，卻可以節省函式呼叫者的時間，還可以將滿頭大汗的他從「試圖精確推斷這個函式可能產生哪種異常」的窘境中解脫出來。

關於 throws 子句的所有這些討論，帶出了一個有趣問題。覆寫（override）某個函式時，你可以從覆寫函式中拋出哪些異常呢？哪些異常應該被放進函式的 throws 子句呢？假設你有這樣的程式碼：

```
import java.io.*;

class Base
{
  public void foo() throws FileNotFoundException
  {
    //...
    throw new FileNotFoundException();
  }
}

class OverrideTest extends Base
{
  public void foo() throws IOException
  {
    throw new IOException();
  }
}
```

編譯它，得到下面結果：

```
OverrideTest.java:14: The method void foo() declared in class
OverrideTest cannot override the method of the same signature
declared in class Base.  Their throws clauses are incompatible.
  public void foo() throws IOException
              ^
1 error
```

這個錯誤顯示，如果某函式拋出一個異常，而該函式又覆寫了其 superclass 函式，那麼它必須受到 superclass 函式所拋異常的型別的約束。覆寫函式所拋出的異常，要不與 superclass 的對應函式具有相同型別，要不必須是這些型別的特化（specializations；譯註：亦即衍生型別）。因此，對於 class `OverrideTest` 的 `foo()`，你可以：

- 不拋出任何異常。
- 拋出 `FileNotFoundException`，或者，
- 拋出 `FileNotFoundException` 的衍生異常（類別）。

如此一來，你就被限制在被覆寫的那個函式的「活動範圍」內了。舉個例子，如果你的覆寫對象（某個 superclass 函式）沒有拋出任何異常，但你的覆寫函式因爲增加了程式碼而可能引發異常，那麼你必須在新函式中捕捉異常並就地處理。你不能將新函式所引發的異常傳播給外界。

實踐 21：使用 `finally` 避免資源洩漏（resource leaks）

Java 異常處理模型與其他語言相比，關鍵字 `finally` 是最出色的新增特性了。`finally` 構件使得某些程式碼總是得以被執行，無論是否發生異常（參見**實踐 16**）。在維護物件內部狀態和清理 non-memory 資源方面，`finally` 尤其適用。如果沒有 `finally`，程式碼會變得錯綜複雜，糾纏不清。例如下面的程式碼就是在沒有 `finally` 的幫助下欲釋放 non-memory 資源而迫不得已採用的寫法：

```
import java.net.*;
import java.io.*;

class WithoutFinally
{
  public void foo() throws IOException
  {
    //Create a socket on any free port
    ServerSocket ss = new ServerSocket(0);
    try {
      Socket socket = ss.accept();
      //Other code here...
    }
    catch (IOException e) {
      ss.close();                                    //1
      throw e;
    }
```

```
      //...
      ss.close();                                    //2
    }
  }
```

這段程式碼建立了一個 socket，並呼叫其 `accept()`。爲避免資源洩漏，必須在退離這個函式之前關閉 socket。於是你在函式的最後一條述句 `//2` 處呼叫 `close()`。然而萬一 `try` 區段內發生異常，情況會怎樣？在這種情形下，程式流程不會到達`//2` 去喚起`close()`。我們必須捕捉異常，並在再次拋出異常之前的 `//1`處放置`close()` 的第二個呼叫句。這便可以確保在退離函式之前關閉 socket。

用這種方式寫碼不僅麻煩，而且容易出錯，但如果你沒有 `finally` 的支援，那也別無選擇（譯註：C++ 就是如此）。不幸的是，在缺乏 `finally` 機制的語言中，程式員也許會忘記以上述方式組織程式碼，導致資源洩漏。Java 的 `finally` 子句解決了這個問題，現在程式碼可以重寫如下：

```
import java.net.*;
import java.io.*;

class WithFinally
{
  public void foo2() throws IOException
  {
    //Create a socket on any free port
    ServerSocket ss = new ServerSocket(0);
    try {
      Socket socket = ss.accept();
      //Other code here...
    }
    finally {
      ss.close();
    }
  }
}
```

無論 `try` 區段是否拋出異常，`finally` 區段都能確保 `close()`被執行起來。於是你可以確信 socket 將被關閉而不會有資源洩漏之虞。上述函式不再需要 `catch` 區段。上一頁第一個例子中的 `catch` 區段主要用來關閉 socket，現在有 `finally` 就足夠了。如果你提供了一個 `catch` 區段，`finally` 區段內的程式碼會在 `catch` 區段完成之後才被執行（參見**實踐 16**）。

finally 區段必須要配合 try 區段或 try/catch 區段來使用。只要 finally 區段存在，它就一定會被執行[5]，絕對不可能在未執行 finally 區段的情況下就退離 try 區段。

實踐 22：不要從 try 區段中回返

try/finally 區段的執行機制十分簡單明瞭，然而一些不恰當的行爲方式甚至會騙過程式員老手的眼睛。是的，只要 finally 區段存在，它就一定會被執行；try 區段的程式碼一離開 try 區段就會進入 finally 區段。造成「程式碼離開 try 區段」的情形包括：

- 拋出異常。
- try 區段正常結束。
- 在 try 區段中執行了 return、break 或 continue 述句，從而引發執行權（execution）離開 try 區段。

現在請你考慮以下程式碼，看看它所引發的後果：

```
class FinallyTest
{
  public int method1()
  {
    try {
      return 2;
    }
    catch(Exception e) {return 3;}
  }

  public int method2()
  {
    try {
      return 3;                                    //1
    }
    finally {                                      //2
      return 4;
    }
```

[5] 這種說法幾乎總是成立。但的確也存在一種辦法，可以在不執行 finally 區段的情況下退離 try 區段。如果在 try 區段中執行 System.exit(0)，應用程式就會中斷運行而不執行 finally。如果你在執行 try 區段的時候拔下計算機的電源插頭，finally 也不會被執行☺。

```
    }

    public static void main(String args[])
    {
      FinallyTest ft = new FinallyTest();
      System.out.println("method1 returns " + ft.method1());    //3
      System.out.println("method2 returns " + ft.method2());    //4
    }
  }
```

執行上述程式碼會讓我們看到兩個結果，一個顯而易見，另一個未必。這段程式碼呼叫兩個函式，`method1()`和 `method2()`，然後列印它們的回傳值。`//3` 呼叫 `method1()`得到結果 2，並列印出來。由於 `method1()`並未拋出異常，所以其 `catch` 區段永遠沒機會執行。`//4` 呼叫 `method2()`得到 4。輸出結果爲：

```
method1 returns 2
method2 returns 4
```

之所以如此，原因在於：無論 `try` 區段發生了什麼事，`finally` 區段都會被執行。本例原本要在 `//1` 回傳 3，但是執行 `return` 句之後，控制權轉移到了`//2` 的 `finally` 區段，引發述句 `return 4`，因此 `method2()`最終回傳整數 4。

程式員傳統上總是以爲，當他們執行 `return` 述句的時候，會立刻離開執行中的函式。但是在 Java 語言中，一旦 `finally` 出現，這種觀點不再是金科玉律。`try` 區段中的 `break` 或 `continue` 述句也可能使控制權進入 `finally` 區段。`finally` 的這種獨特性質有可能令人困惑，陷入漫長的除錯困境而無法自拔。

爲了繞離這個陷阱，請不要在 `try` 區段中發出對 `return`、`break` 或 `continue` 述句的呼喚。萬一無法避免，一定要確保 `finally` 的存在不會改變你的函式的回傳值。這類問題最容易出現在程式碼維護時期，即使仔細的設計和實作也不能完全避免。優秀的註釋和細心的程式碼復審（code reviews），可以使你避開這類問題。

實踐 23：將 try/catch 區段置於迴圈之外

異常（exceptions）可能對你程式碼效率（performance）產生負面影響。至於是否真的產生這種影響，與你的程式碼組織大有關係，也與 JVM 是否在執行期使用 JIT 編譯器進行程式碼最佳化有關。關於異常和效率，我們需要從以下兩個方面加以考慮：

- 異常被拋出後所帶來的影響。
- 程式碼中 try/catch 區段的影響。

拋出異常並不是一種無需代價的動作。「Java 異常」說到底是個什麼東西呢？它們是物件，而物件需要被創建。創建物件所花費的成本並不便宜（參見**實踐 32**），於是拋出異常也就需要一些代價。欲拋出一個異常，你得撰寫這樣的述句：

```
throw new MyException();
```

這便創建了一個新物件，然後控制權轉移到 catch 區段或 finally 區段，甚至轉移到呼叫端。由於拋出異常需要承擔一些代價，所以你應當只對出錯情況使用異常。如果將異常用於流程控制，程式碼無法像常規流程結構那樣高效和清晰（參見**實踐 24**）。請將異常的使用限制在「出錯」場合和「失敗」場合。是的，你會希望程式碼運行時健步如飛，但你通常不在意它失敗時要花掉多長時間。

當我們評量 Java 效率的時候，必須將 JVM 及其所在的操作系統一併考慮進去。人們已經注意到，即使執行同一份程式碼，不同的 JVMs 之間的執行時間也有所不同。因此你得對你的系統作某種程度的評測（profiling），判定效率上的差異。**實踐 2**3 所產生的效率資料，是在本書 p98 詳細描述之硬體和軟體配置下得出的。

將 try/catch 區段放在迴圈之內，會減慢程式碼的執行速度，以下程式碼可以證明這一點：

```
class ExcPerf
{
  public void method1(int size)
  {
    int[] ia = new int[size];
    try {
      for (int i=0; i<size; i++)
        ia[i] = i;
    }
```

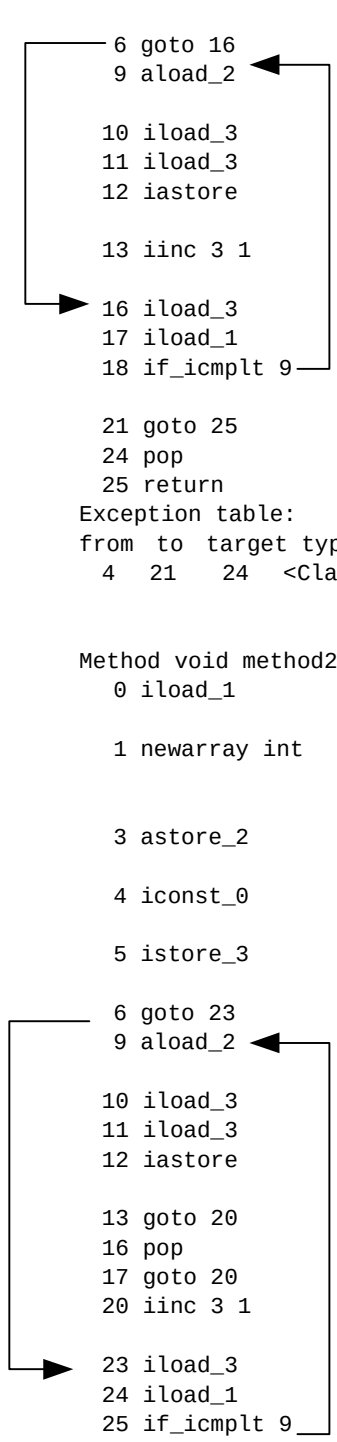
```
      catch (Exception e) {} //Exception ignored on purpose
    }

    public void method2(int size)
    {
      int[] ia = new int[size];
      for (int i=0; i<size; i++)
      {
        try {
            ia[i] = i;
        }
        catch (Exception e) {}  //Exception ignored on purpose
      }
    }
  }
```

method1()和 method2()內含幾乎一樣的程式碼。method1()在迴圈之外有個 try/catch 區段，method2()則將 try/catch 區段置於迴圈內。請注意，兩個函式都沒有拋出異常。他們只是使用 try/catch 區段將一些程式碼包圍起來。

在「啓用 JIT 編譯器」的情形下執行上述程式碼，兩個函式的執行時間並沒有什麼不同。可是一旦你將 JVM 的 JIT 關閉再執行之，method2()大約比 method1()慢 21%。

這或許會令你以爲，兩個函式產生的bytecode一定大不相同。事實上，除了method2()內含少量「用於迴圈內部所含之 try/catch 區段」的操作碼（opcodes）外，二者幾乎完全相同。當 JIT 編譯器啓動，程式碼被最佳化，消除了二者執行期的差異。然而如果沒有 JIT，每次迴圈迭代（iteration）都得花費一些額外代價。下面就是兩個函式所產生的 bytecode：

```
Method void method1(int)
   0 iload_1          //Push the value stored at index 1 of the
                      //local variable table(size) on the stack.
   1 newarray int     //Pop the size parameter and create a new
                      //int array with size elements. Push the
                      //newly created array reference(ia).
   3 astore_2         //Pop the array reference(ia) and store it
                      //at index 2 of the local variable table.
   4 iconst_0         //Beginning of try block. Push 0 for the
                      //initial value of the loop counter(i).
   5 istore_3         //Pop 0(i) and store it at index 3 of the
```



```
                        //local variable table.
    6 goto 16           //Jump to location 16.
    9 aload_2           //Push the object reference(ia) at index 2
                        //of the local variable table.
   10 iload_3           //Push the value at index 3(i).
   11 iload_3           //Push the value at index 3(i).
   12 iastore           //Pop the top three values. Store the value
                        //of i at index i in the array(ia).
   13 iinc 3 1          //Increment the loop counter(i) stored at
                        //index 3 of the local variable table by 1.
   16 iload_3           //Push the value at index 3(i).
   17 iload_1           //Push the value at index 1(size).
   18 if_icmplt 9       //Pop both the loop counter(i) and size.
                        //Jump to location 9 if i is less than size.
   21 goto 25           //End of try block. Jump to location 25.
   24 pop               //Beginning of catch block.
   25 return            //Return from method.
 Exception table:       //If a java.lang.Exception occurs between
 from  to  target type  //location 4(inclusive) and location
   4   21    24   <Class java.lang.Exception>
                        //21(exclusive) jump to location 24.

 Method void method2(int)
    0 iload_1           //Push the value stored at index 1 of the
                        //local variable table(size) on the stack.
    1 newarray int      //Pop the size parameter and create a new
                        //int array with size elements. Push the
                        //newly created array reference(ia).
    3 astore_2          //Pop the array reference(ia) and store it
                        //at index 2 of the local variable table.
    4 iconst_0          //Push 0 for the initial value of the loop
                        //counter(i).
    5 istore_3          //Pop 0(i) and store it at index 3 of the
                        //local variable table.
    6 goto 23           //Jump to location 23.
    9 aload_2           //Beginning of try block. Push the object
                        //reference(ia) at index 2.
   10 iload_3           //Push the value at index 3(i).
   11 iload_3           //Push the value at index 3(i).
   12 iastore           //Pop the top three values. Store the value
                        //of i at index i in the array(ia).
   13 goto 20           //End of try block. Jump to location 20.
   16 pop               //Beginning of catch block.
   17 goto 20           //Jump to location 20.
   20 iinc 3 1          //Increment the loop counter(i) stored at
                        //index 3 of the local variable table by 1.
   23 iload_3           //Push the value at index 3(i).
   24 iload_1           //Push the value at index 1(size).
   25 if_icmplt 9       //Pop both the loop counter(i) and size.
```

```
                    //Jump to location 9 if i is less than size.
  28 return         //Return from method.

Exception table:    //If a java.lang.Exception occurs between
from  to  target type //location 9(inclusive) and location
  9   13    16   <Class java.lang.Exception>
                    //13(exclusive) jump to location 16.
```

箭頭顯示兩個函式的迴圈結構。基於這些差異，良好的編程習慣是將 `try/catch` 區段置於迴圈之外。你的客戶或許使用無 JIT 能力的 JVM，或是由於記憶體的考慮而在執行期將 JIT 關閉，因此你不可以假設迴圈內的 `try/catch` 區段不會對程式效率帶來負面影響。

如果你想在不啓動 JIT 的情況下執行名爲 `Test.class` 的檔案，請下這個命令：

```
java -Djava.compiler=NONE Test
```

這會使得程式 `Test` 執行時，JVM 的 JIT 能力被抑制。

實踐 24：不要將異常（exceptions）用於流程控制

「使用異常來控制流程」是一個糟糕的主意。運用 Java 的標準語言構件（standard language constructs）來表達程式流程，會更加清晰。考慮下面的程式碼：

```
class DoneWithLoopException extends Exception
{}

class Test
{
  public void foo()
  {
    //...
    try {
      while(true)
      {
        //Do something...
        if (some loop terminating condition)
          throw new DoneWithLoopException();
      }
    }
    catch (DoneWithLoopException e)
    {}
    //...
  }
}
```

這段程式碼運用異常（exceptions）來控制流程。它沒有使用正常的迴圈終止條件，而是使用異常來中斷迴圈。這個程式碼可以運作，但效率低下，含義模糊，而且難以維護。如果你看到這樣的程式碼，請當機立斷重新寫過。請記住，「異常處理」（exception handling）只用於異常情況，不要把它拿來用於流程控制貫穿你的程式。

實踐 25：不要每逢出錯就使用異常（exceptions）

異常處理機制（exception handling）的設計初衷，是作爲傳統的錯誤處理技術（error handling techniques）的一份更強固的替代品。於是有些程式員以爲，異常處理應當用在所有出錯情形中，並且應該極力避免使用傳統的錯誤處理程序。

人們可能會濫用異常處理。事實上它不應當被用於流程控制（參見**實踐 24**），或是被用以揭發「非錯誤」情況。「異常處理」應當和「傳統錯誤處理技術」兩相配合，共同建立高效、易於理解的程式碼。

下面是兩種風格的對比，第一種使用傳統錯誤處理技術，第二種對所有錯誤情形都使用異常。首先使用傳統的回傳碼（return code）進行錯誤檢查：

```
int data;
MyInputStream in = new MyInputStream("filename.ext");
data = in.getData();
while (data != 0)                                    //1
{
  //Do something with data
  data = in.getData();
}
```

注意，`//1` 檢測 `getData()`回傳值是否不爲 0。如果是 0，就可以假設你位於 stream 尾部，於是便可以中斷迴圈。這樣做不但有效而且直觀。接下來，改用異常處理的嚴厲姿態修改這個程式碼。這時候不再像前面那樣依賴回傳值，而是改用異常。以下程式碼在原先的基礎上增加了異常處理：

```
int data;
MyInputStream in = new MyInputStream("filename.ext");
```

```
try {                                                    //1
   while(true)
   {
      data = in.getData();
      //Do something with data
   }
}
catch (NoMoreDataException e1) {}
```

注意，`//1` 的 `try/catch` 區段將內含 `getData()`的迴圈包圍了起來。這一次 `getData()`於 stream 爲空的時候不再回傳 0，而是拋出 `NoMoreDataException`。儘管這段程式碼比原先那個看起來「醜陋」些，有些程式員還是認爲程式碼就應當這樣寫。他們主張，如果你能使用異常，就不該再使用傳統的錯誤處理技術。

但是請你注意，未使用異常的那一段程式碼，儘管用的是舊式方法來處理「錯誤」或「預期外的結果」，卻更直觀淺白。異常可能被濫用，就像上述第二個例子那樣。

請你在面對「出乎程式可預料」的行爲時，才使用異常。先前的例子中，你已經預料到會到達 stream 尾部，因此 `getData()`回傳一個簡單的 0 是合適而且自然的。在此處拋出異常並不明智，因爲這不是一個異常（exceptional）情形，而是可以預料到的。然而你不會預料到 stream 被破壞，那種情況下你就必須產生一個異常。重點是，不要針對所有情形使用異常，應該將異常用於符合其意義的地方，也就是出現了「非尋常情況」的地方。

一味地、獨佔性地使用異常（就像上述第二個例子那樣），或許會使你的程式在處理異常方面顯得「純潔」，但卻違背了「不要每逢出錯就使用異常」的原則。「簡單地回傳一個 0」比起「產生一個異常物件，並要求呼叫者實作 `catch` 區段以便處理那個異常」更快捷，更直觀。

實踐 26：在建構式（constructors）中拋出異常

由於建構式沒有回傳值，想要從建構式獲得傳統的錯誤報告（error reporting）就很成問題。一旦建構式失敗，你無法回傳一個錯誤碼。的確，建構式不是函式，但這並不能阻止你在建構式中拋出異常。

想要從建構式中得到錯誤報告，有數種技術可以辦到。技術之一就是雙階段建構（two-stage construction），將有可能產生錯誤的程式碼移出建構式，各自組成函式，這些函式可以回傳錯誤碼。這個技術的缺點是 class 用戶必須先呼叫建構式，然後呼叫那些可能產生錯誤的程式碼，然後再檢查回傳碼或捕捉異常。

雙階段建構的另一種形式，是對「待建之物」使用一個內部旗標（internal flag）。這個旗標代表此一物件被建構後的有效性。如果建構式毫無差錯地順利完成任務，它就在物件內設置一個旗標，表示這個物件一切就緒，可安心使用。如果建構式因爲某種原因而失敗，就設置旗標表示這個物件處於無效狀態，不能被安全地使用。

你可以採用以下兩種方法的任一種來使用這個內部旗標。第一種方法是，物件用戶在呼叫物件的任何函式之前，必須先呼叫其 class 的某個函式，檢查此一旗標是否有效。第二種方法是，物件的每個函式一開始都先呼叫上述函式以確認物件的有效性。如果物件處於無效狀態，就回傳一個錯誤碼或拋出一個異常。這麼做可確保使用者不會對著一個無效的物件執行函式。

然而這些技術都缺少所謂的強固性（健壯性，robustness）。這些技術要求 class 使用者不得忘記執行上述的「兩階段建構」，不得忘記呼叫某個特別函式來測試物件的有效性 — 每個函式都必須先呼叫該特別函式，於是因爲「檢查內部狀態旗標」而帶來額外開銷（overhead）。

藉由「在建構式中拋出異常」，你可以避免所有這些問題。考慮下面的程式碼：

```
import java.io.*;

class Foo
{
  public Foo (String fileName) throws FileNotFoundException,
                                      IOException
  {
    FileReader fr = new FileReader(fileName);// 譯註：可能引發異常
    BufferedReader br = new BufferedReader(fr);
    String str = br.readLine();              // 譯註：可能引發異常
    //...
  }
  public void scanfile() {}
}
```

```
class Test
{
  public static void main(String args[])
  {
    Foo somefoo = null;
    try {
      somefoo = new Foo("temp.fil");
    }
    catch (FileNotFoundException fnfe) { //Catch constructor
      //...                               //failure
    }
    catch (IOException ioe) {  //Catch constructor failure
      //...
    }
    //Use somefoo
    somefoo.scanfile();
  }
}
```

class Foo 建構式接受一個 String 引數，這個 String 代表一個檔名。然後它嘗試打開這個檔案。如果檔名無效，建構式就引發一個 FileNotFoundException 異常。如果檔名有效，建構式便試圖讀取它。但讀取這個檔案另有可能引發一個 IOException 異常。由於這個建構式不處理這些異常（譯註：因此會向外傳播），它得在 throws 子句中列出它們（參見**實踐 19** 和**實踐 20**）。

在 main()函式中，try/catch 區段將 Foo 建構式包圍起來，以便處理任何可能的失敗。這個方法具有一個額外好處：倘若建構式拋出異常，並且呼叫端忽略處理這個異常，區域變數 somefoo 會被設成 null。這麼安排是因爲它既然沒有得到適當的建構，也就不可能被安全運用。如果這種情況下你試圖存取變數 somefoo，會導致 Java 執行期產生一個 NullPointerException 異常。

儘管建構式不是一般函式，它們仍然可以引發異常，並支援 throws 子句。以這種方式來處理建構失敗，是最強固、最高效的選擇，從 class 用戶的角度來看，它要求的手工干預最小。

實踐 27：拋出異常之前先將物件恢復為有效狀態（valid state）

拋出異常很容易，困難的是如何將拋出異常所導致的損害減至最小。常見的情況是，程式員被誤導，以爲關鍵字 try、throw、catch、throws 和 finally 就是錯誤處理的起點和終點。其實它們只是正確處理錯誤的起點而已。

問問你自己，拋出異常是爲了什麼？明顯的目的是將已發生的問題通知系統的其他部分。隱含的目的則是讓軟體捕獲異常，使系統有可能從問題中抽身回復（recover）而維持正常運行狀態。如果異常被引發後，系統回復並繼續運轉，引發異常的那一段程式碼可以被「復入」（reentered）。那些回復至「異常拋出前之狀態」的物件，在系統繼續執行期間可以再被使用。令人迷惑的是，拋出異常之後，這些物件的狀態究竟如何？系統還可以正確運轉嗎？

如果你的物件處於一種無效（*invalid*）狀態或未定義（*undefined*）狀態，拋出異常又有什麼用呢？如果你的物件處於不良（*bad*）狀態，那麼異常回復後的程式碼還是很可能失敗。因此拋出異常之前，你必須考慮物件當時處於什麼狀態。如果它們的狀態會使得「即使異常回復，程式碼仍然失敗」，你就得在拋出異常之前，考慮如何讓它們處於有效狀態。

通常，程式員會假設函式中的程式碼將毫無錯誤地完成工作。一旦錯誤發生，原先假設的部分甚至全部便都靠不住了。考慮下面的程式碼：

```
class SomeException extends Exception
{}
class MyList
{}
class Foo
{
  private int numElements;
  private MyList myList;

  public void add(Object o) throws SomeException
  {
    //...
    numElements++;                                    //1
    if (myList.maxElements() < numElements)
    {
      //Reallocate myList
      //Copy elements as necessary
      //Could throw exceptions
    }
    myList.addToList(o);  //Could throw exception     //2
  }
}
```

這段程式碼包含了一個 add()，用以將一個物件加入 list 中。這個函式首先在 //1 將一個計數器值累加 1，該值記錄了 list 中的物件數量。然後它有條件地重新配置 list，並在 //2 爲 list 添加一個物件。這個程式碼有嚴重缺陷，是的，請注意 //1 和 //2 之間可能拋出異常。如果 //1 身後拋出異常，那麼 Foo 物件就處於無效狀態，因爲計數器 numElements 的值不對。如果函式呼叫者從拋出的異常中回復過來，並再次呼叫這個函式，由於 Foo 物件處於無效狀態，很可能發生其他問題。

修正這個問題很是容易：

```
class Foo
{
  private int numElements;
  private MyList myList;

  public void add(Object o) throws SomeException
  {
    //...
    if (myList.maxElements() == numElements)          //1
    {
      //Reallocate myList
      //Copy elements as necessary
      //Could throw exceptions
    }
    myList.addToList(o);  //Could throw exception
    numElements++;                                    //2
  }
}
```

爲了修正錯誤，只要修改 //1 的測試行爲，並將 numElements 的累加動作移到函式尾部的 //2 處，這樣就確保計數器 numElements 永遠準確，因爲你在成功地將物件添加到 list 之後才增加計數器值。這是一個簡單範例，但它揭示了一個潛在的嚴重問題。

你所放心不下的，肯定不僅僅是「執行中的那個物件」，還包括「被我們關注的那個函式修改過的物件」。舉個例子，異常發生時，或許函式內已經完全建立或部分建立了一個檔案，或打開了一個 socket，或是對另一台計算機發出了遠端呼叫（remote method calls）。如果你希望你的程式碼被復入（reentered）後還能正常運轉，你需要瞭解所有受到影響的物件的狀態，以及系統本身的狀態。考慮以下程式碼：

```
import java.io.IOException;

class MutualFund
{
  public void buyMoreShares(double money)
  {}
  //...
}

class Customer
{
  private MutualFund[] fundArray;

  public Customer()  {}

  public MutualFund[] funds()
  {
    return fundArray;
  }

  public void updateMutualFund(MutualFund fund) throws
    DatabaseException
  {}

  public void writePortfolioChange() throws IOException
  {}
  //...
}

class DatabaseException extends Exception
{}

class Services
{
  public void invest(Customer cust, double money) throws
    IOException, DatabaseException
  {
    MutualFund[] array = cust.funds();                    //1
    int size = array.length;

    for (int i=0; i<size; i++)
    {
      ((MutualFund)array[i]).buyMoreShares(money);          //2
      cust.updateMutualFund(array[i]);
      cust.writePortfolioChange();
    }
    //...
  }
}
```

在這個實例中，`invest()`將客戶（*customer*）的資金投資到客戶的各個互惠基金（*mutual fund*）的帳戶（*accounts*）中。`Customer` class 內含一個 `funds()`，回傳客戶持有之全部基金(*funds*)所組成的一個 arrays。`invest()`透過 `buyMoreShares()` 對客戶的每個基金買進相同金額，然後更新資料庫內的 `MutualFund` 物件，並修改 `Customer` 物件的有價證券（*portfolio*）資訊。對有價證券的修改動作會建立起一筆帳目活動記錄（*account activity*），讓客戶得知交易（*transaction*）情況。迴圈中的最後兩個函式有可能失敗並拋出異常。由於 `invest()`並不處理異常，爲避免編譯錯誤，乃將這些異常列入 `throws` 子句中。

某些程式員寫出了這樣的程式碼，測試後確定運轉正常，於是以爲萬事大吉。測試這段程式時，他們發現，如果其中某個函式失敗，會拋出一個相應異常，然後如預期般地退離函式。如果他們的測試不夠徹底，很可能不會發現潛伏的問題。

每當你撰寫可能引發異常的程式碼時，都必須問自己：(1) 異常發生之時、(2) 異常處理過後、(3) 復入（reentered）這段程式碼時，會發生什麼事情？程式碼運轉還能夠正常嗎？在這個例子中，程式碼確實存在一些問題。

假設某位客戶在三個互惠基金中擁有股票（*shares*），並打算爲每個基金各自再買進$1,000。`//1` 取得了所有基金構成的 array，`//2` 爲 array 的第一筆基金購買 $1,000 的股票，並透過 `updateMutualFund()`，成功地將更新後的基金寫入資料庫。然後呼叫 `writePortfolioChange()`，將某些資訊寫入檔案。這時候這個函式失敗了，因爲磁碟空間不足，無法建立檔案。於是在對第一個 `MutualFund` 物件完成「完整的三個步驟」之前，拋出一個異常，意外而魯莽地退離了 `invest()`。

假設 `invest()`呼叫者透過釋放磁碟空間等手法，順利處理了這個異常，而後再次呼叫 `invest()`。當 `invest()`再次執行時，`//1` 取得基金 array，而後進入迴圈，爲每支基金購買股票。但是不要忘了，先前第一次呼叫 `invest()`的時候，你已經爲第一支基金購買了$1,000 的股票。此時又再做一遍。如果這次 `invest()`順利爲客戶的三筆基金完成了操作，你就是爲第一筆基金購買了$2,000，其他兩支基金各購買$1,000，這是不正確的。顯然，這不是你期望的結果。

就「首先釋放磁碟空間，而後再次呼叫這個函式」的做法而言，你正確地處理了異常。你沒有做的是，在拋出異常之後、退離 invest()之前，關注物件的狀態。

此問題的一個修正辦法是，在 MutualFund 和 Customer 兩個 classes 中添加函式，並在 invest()處理異常時呼叫它們。這些函式用來撤銷未竟全功的一些事件。invest()必須增加 catch 區段，處理任何被引發的異常。這些 catch 區段應該呼叫相應函式，重置（重新設定）物件狀態，這麼一來如果這個函式下次再被呼叫，就可以正確執行了。修改後的程式碼看起來是這樣：

```
import java.io.IOException;

class MutualFund
{
  public void buyMoreShares(double money)
  {}
  public void sellShares(double money)
  {}
  //...
}

class Customer
{
  private MutualFund[] fundArray;

  public Customer()
  {}

  public MutualFund[] funds()
  {
    return fundArray;
  }

  public void updateMutualFund(MutualFund fund) throws
      DatabaseException
  {}
  public void undoMutualFundUpdate(MutualFund fund)
  {}

  public void writePortfolioChange() throws IOException
  {}
  //...
}

class DatabaseException extends Exception
{}
```

```
class Services
{
  public void invest(Customer cust, double money) throws
    IOException, DatabaseException
  {
    MutualFund[] array = cust.funds();
    int size = array.length;

    for (int i=0; i<size; i++)
    {
      ((MutualFund)array[i]).buyMoreShares(money);
      try {
        cust.updateMutualFund(array[i]);
      }
      catch(DatabaseException dbe)  //Catch exception and return
      {                             //object to a valid state.
        ((MutualFund)array[i]).sellShares(money);
        throw dbe;
      }
      try {
        cust.writePortfolioChange();
      }
      catch(IOException ioe)  //Catch exception and return object
      {                   //to a valid state.
        ((MutualFund)array[i]).sellShares(money);
        cust.undoMutualFundUpdate(array[i]);
        throw ioe;
      }
    }
    //...
  }
}
```

現在，即使 `invest()`拋出異常，而後復入（reentered）`invest()`，上述程式碼也能一如期望地正確執行了。爲了讓這段程式碼正常運轉，`MutualFund` 和 `Customer` classes 各增加一個函式，用以在發生異常時取消（回復, *undo*）已對物件進行的操作。`invest()`內的迴圈也有了修改，捕捉可能發生的異常，並重置（*reset*）物件狀態。在物件成爲有效狀態、確保復入（reentered）時可正確運轉後，重新拋出異常，這樣呼叫端便可嘗試回復（recovery）。

這段程式碼與原先版本相比，較不易撰寫和遵循，但對於強固的、可從異常狀態中回復的程式碼而言，這些步驟必不可少。

另一個需要考慮的問題是，萬一「回復函式」`undoMutualFundUpdate()`和

`sellShares()`也失敗了，情況會變得如何？如果要保持系統順利運轉，你也需要對付這些失敗。的確，正如你所看到，在拋出異常的時候保持物件處於有效狀態，可能非常困難。

如果希望呼叫者從你所引發的異常中回復，你必須察看所有可能發生異常的地方。如果函式在異常發生處不顧一切地退離（exit），那麼你有必要檢查這種行爲是否使你的物件處於以下狀態：「復入這個函式時可產生正確結果」。如果沒有令物件處於有效狀態，你就必須採取必要措施，確保重新進入函式時得以按我們所期望的方式正常運轉。

本實踐內容所表述的過程和所謂的「交易」（transaction）十分類似。進行交易時，你必須有一個「提交+回復」（commit and rollback）計劃。如果交易所牽涉的每一部分無法全部正確完成，就必須取消整筆交易。你也許會考慮以一個全域性交易（global transaction）或一個「提交+回復」策略來實作解決方案。

不幸的是，解決這類問題十分困難和耗時。欲正確實作這些解決方案，不僅得花費大量編程時間和測試時間，還需要大量的艱苦思考。但如果你在編程之後才開始忙碌這些問題，那可比在設計和編程初始就將這些問題銘記於心要棘手得多 — 你將因此花費更多的工作和努力！

附錄

如何學習 Java

一知半解猶如盲人瞎馬
A little learning is a dangerous thing.
—Alexander Pope

什麼是學習和擴展 Java 知識的最佳途徑？我從不自詡知道最佳方法，但我確實知道一種非常有效的方法，可以增強你的 Java 開發技巧。我曾在 IBM 開設一門課程，後來衍化爲一個高效的學習工具。

數年前我受邀設計一種方法，指導 IBM 實驗室的其他程式員學習 C++。當時我正參與一個非常龐大的 C++ class library 開發專案。我說服另外三位 C++ 專家加入，規劃了一個名爲「C++ 系列研討班」的課程。這個系列課程非常成功，以致於後來當我們的焦點由 C++ 轉移至 Java 時，我們將這個課程演化爲「Java 系列研討班」。

這個系列研討班的運作方式如下。由三至四位精通 Java 的人員 — 最好是專家 — 組織並主持這個小組，導師（扮演良師益友的角色）建立一個關於 Java 的興趣主題列表。小組的每一名成員必須從列表中選擇一個主題。小組成員可以自行推薦主題，但需得到導師的批准。然後導師針對這些主題建立一個時程表，指派每一次報告的講員和時間。爲了給講員充足的準備時間，導師領銜做最初數週報告。這樣也爲後來的報告定下基調和風格。

每位成員對於被指派的主題分頭進行研究，並且在約定時間向小組的其他成員報告。我們發現，每次報告 90 分鐘就夠了。小組應當以一個週期性的時程表爲基礎

來運轉。根據我們的經驗，一週一次很合適，可以保持小組的平穩發展。我們還有一位贊助人，爲每次會議提供點心（這是對出席者的一個額外獎勵和驚喜）。

小組會議應當儘可能適意些，研討範圍限於小組成員所報告的內容。我們不要求任何人站著，或使用標準的簡報圖表。無論什麼樣的形式和做法，只要講員認爲可以最有效地展現主題，都是被允許的。

對某些人而言，向陌生的一群人報告陌生的主題，可能會感到緊張。爲了免除這種麻煩，我們將聽眾限制爲參加報告的人，這樣的感受可能會好一些。每個人都認識房間中的其他人，這有助於他（或她）專心報告。

導師最好事先瞭解報告內容。這樣他們就可以在報告期間提出一些相關問題，幫助現場保持活力。這樣做的目的不是要令任何人難堪，而是使談論活躍起來，並激勵參與者儘可能學習報告的相關內容。

即使講員無法招架現場問題，也無所謂。每個人都知道，講員只有有限的時間準備講題，並非是一方專家。這種情況下，講員應該將問題記錄下來，並於報告結束後以 e-mail 形式向其他小組成員發送他所找到的答案。

如果報告採電子形式，文件和範例程式碼應置於任何人都可以取用的網站上。這個網站還應當包含其他相關資訊，以及 Java 資源的連接（links）。

加入系列研討班的條件是：選一個主題向大家報告。收益則是從「出席其他報告和準備自己的報告」中獲得學習。我們要求每一位參與者至少具有 Java 使用經驗。我們不將重點放在介紹性的材料上，而是中級至高級的專題。

小組人數決定這個系列研討班的持續時間。過去，我們最少 10 人，最多達到 40 人。太大的小組會使系列研討班的時間拉得過長。合適的人數是 12~20。這樣的規模小到足以輕鬆交流，並於 15~25 周內結束（在每週一次會議的前提下）。

總的說來，這種研討形式在北卡羅萊那（North Carolina）的 IBM Research Triangle Park 獲得了巨大成功。我們數年前舉辦了兩期 C++ 系列研討班，最近完成了第二期 Java 系列研討班。隨著興趣的擴散，每一次參加的人都愈來愈多。

我鼓勵你在你的工作場所設立這樣一個課程。這是一個極好的機會，讓一群人共同學習 Java。而你也因此有機會與原本無緣合作的同仁共事。

進階讀物

Further Reading

無論走到哪裡，我都被問道：我是否認爲大學扼殺了作家。
我的看法是，大學不足以扼殺他們。
Everywhere I go I'm asked if I think the university stifles writers.
My opinion is that they don't stifle enough of them.
—Flannery O'Connor

你到哪裡尋求關於 Java 和軟體工程的權威解答呢？快速瀏覽一下書店和你喜愛的網站，你會發現並不缺乏 Java 書籍。問題在於，什麼樣的書籍值得擁有？什麼樣的書籍爲程式員提供了以 Java 做爲開發環境時的必備資訊？以下清單無論如何也稱不上完整，不過它包含了我所參閱的書籍和期刊。我完整閱讀了它們（或至少它們最具代表性的那一部分），所以我能給出一份嚴謹的推薦。

The Java™ Programming Language, Second Edition, Ken Arnold and James Gosling, Addison-Wesley, 1998, ISBN 0-201-31006-6.

這本書是 Java 語言基礎的一個良好資源。某些情況下，它對某些主題甚至提供了比 *The Java Language Specification* 更詳盡的解說。總的說來，這是一本優秀的教本，擁有令人信服的討論和清晰的範例。

Practical Java

The Java™ Language Specification, James Gosling, Bill Joy, and Guy Steele, Addison-Wesley, 1996, ISBN 0-201-63451-1.

如果你想查閱 Java 語言的權威資訊和解答，看這本書就對了。本書稱得上是「語言精鑽者（language lawyer）的夢想」，回答了 Java 語言細節的一切問題。

The Java™ Virtual Machine Specification, Second Edition, Tim Lindholm and Frank Yellin, Addison-Wesley, 1999, ISBN 0-201-43294-3.

本書包含 JVM 細節的準確資訊和解答。爲了理解如何建立 JVM、JVM 如何運作、如何移植一個既有的 JVM、如何撰寫你自己的 JVM，本書都是必讀之物。它還講解了 JVM 指令集（bytecode），是編譯器和 JVM 撰寫者的重要讀物。它也提供了關於諸如「執行緒的私有和專用記憶體、物件初始化和同步機制」等專題的詳細解說。

撰寫《*Practical Java*》期間，以上三本書從不離開我的左右。我反反覆覆地參閱它們，一絲不苟地鑽研它們。我無法想像，如果沒有它們，我如何接受一項重大的 Java 開發任務。

The Practice of Programming, Brian W. Kernighan and Rob Pike, Addison-Wesley, 1999, ISBN 0-201-61586-X.

儘管並非專爲 Java 而作，這本書卻爲優秀的編程實踐提供了與語言無關的建議和評論。作者將數十載編程經驗濃縮成一本及時的好書，不論你使用任何編程語言，都可以從中受益。

Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995, ISBN 0-201-63361-2.

範式（patterns）經典之作。每一位軟體設計者和實作者都應該閱讀此書。

Design Patterns CD: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1998, ISBN 0-201-63498-8.

《*Design Patterns*》的 CD-ROM 版本。

Writing Efficient Programs, Jon L. Bentley, Prentice Hall, 1982, ISBN 0-13-970251-2, or 0-13-970244-X (paperback).

儘管此書已經絕版，卻值得你花時間找一本來閱讀。雖然其中大部分程式範例採用 Pascal 撰寫而顯得有些過時，其觀念依然深具價值。Bentley 對程式碼的效率提供了中肯的建議和分析。

Programming Pearls, Second Edition, Jon L. Bentley, Addison-Wesley, 2000, ISBN 0-201-65788-0.

這是一本出色的小品合集，彙集了編程問題和解決方案。此書有一個章節專門探討效率（performance）。

Java Report, SIGS Publications, New York, NY.

這本一流的 Java 開發雜誌內容非常出色，而且愈來愈優秀。它涵蓋了 Java 的方方面面，從廣泛的業界資訊，到詳盡的編程文章。最近這本雜誌正以更多篇幅關注內容詳盡的編程文章，探討 Java 開發技術的各個層面。

索引
Index

當思考衰退，至少還有文字在手。
When ideas fail, words come in very handy.
— Johann Wolfgang Von Goethe

注意：圓括號內的數字代表本書的實踐（PRAXIS）編號。如果沒有這個號碼，表示該索引條目出現於實踐文字之外，例如在第一章之前的序言資訊中。

B

C

D

E

F

G

H

I

M

N

P

Q

R

S

T

U

V

W

Z