GOTOP

Effective Java

Programming Language Guide

中文版

PEARSON Education Taiwan

碁峰

柯建 譯

Effective Java

Programming Language Guide

中文版

Joshua Bloch 著

侯捷 譯

To My family: Cindy, Tim, and Matt

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,

[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。）

Cloneable, Serializable, Equality, Immutability, Multithreading (Synchronization), 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。

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）。

※ 本書儘量保留與 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。"**user**" 譯爲用戶，"**client**" 譯爲客戶（見 p.4）。

※ "**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 "**destroy**" 譯爲「銷毀」。動詞 "**refer**" 譯爲「指涉、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」。動詞 "**build**" 譯爲構築。

※ 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。"**constructor**" 譯爲建構式。

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

※ 本書將動詞 "**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）。"clone" 做爲名詞則譯爲「克隆件」，"copy" 做爲名詞則譯爲「副本」或「複件」。

※「*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，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，並儘可能於突兀處或不易表達處中英並陳。

-- 侯捷

目錄

Contents

序文

Foreword

如果一位同事這樣對你說："Spouse of me this night today manufactures the unusual meal in a home. You will join?"（我的配偶今晚在家裡製作了一頓非比尋常的晚餐，你要不要加入我們？）你的腦子裡可能閃過三個念頭：(1) 你受邀參加一頓晚餐，(2) 你的同事的母語不是英文，(3) 好一個謎語。

如果你曾經學習第二外國語，並嘗試在課堂外使用它，你應該清楚有三件事必須掌握：語言的結構（語法, grammar）、事物的正確說法（語彙, vocabulary）、如何以慣用而有效的方式表述日常事物（語用, usage）。課堂上往往只涵蓋前兩點，至於第三點，在努力使對方明白的同時你經常會發現當地人對你的表述忍俊不禁。

程式語言也是如此。你需要理解語言的核心 — 它屬於演算型的（algorithmic）、函式型的（functional）還是物件導向的（object-oriented）？你需要熟悉語彙 — 標準程式庫提供了什麼樣的資料結構、操作和功能？你還需要熟悉慣用而有效的程式構築方式。程式語言方面的書籍一般都只涵蓋前兩點，對第三點只做蜻蜓點水式的提及。這可能是因爲前兩點較易編寫，畢竟「語法」和「語彙」僅僅是語言本身的屬性，「語用」卻是這門語言的使用社群（community）的共同特徵。

舉個例子，Java 是物件導向程式語言，實現單一繼承（single inheritance），並在每個函式（method）內支持強制的（以述句爲導向的, statement-oriented）撰碼風格。Java 標準程式庫對於圖形顯示、網絡、分佈式運算（distributed computing）和安全防護（security）都提供支援。但現實生活中如何才能對它做最佳運用呢？

還有一個重點。程式和口頭用語以及大部份書籍雜誌不同的一點是，它會隨著時

間而改變。僅僅只是製造出一堆能夠有效運作並且易於理解的程式碼，通常是不夠的，你還必須把程式碼組織成一種易於修改的形式。面對某個任務 T，可能有 10 種程式撰寫方式，其中 7 種可能是笨拙、低效或難以理解的。剩下的 3 種，哪一種最可能接近下一年度新版本呢？

目前已有大量書籍供你學習 Java 語法，包括《*The Java Programming Language*》by Arnold, Gosling, and Holmes [Arnold00]，以及《*The Java Language Specification*》by Gosling, Joy, and Bracha [JLS]。同樣地，關於 Java 程式庫和 Java API 的書籍也有許多。

本書的定位是滿足你的第三個需求：普遍及高效的 Java 用法。作者 Joshua Bloch 在 Sun Microsystems 公司從事 Java 語言的擴展、實現和使用已有多年，他還大量閱讀了他人撰寫的程式碼。他在本書之中提出優秀的、經過系統組織的建議，這些建議主要用在如何架構你的程式碼使之有效運作、易被他人理解，並於日後改動和升級時比較不會帶來麻煩，甚至讓你的程式更加令人愉悅、更加優美、更爲雅致。

Guy L. Steele Jr.
Burlington, Massachusetts
April 2001

前言

Preface

1996 年，我打點行囊，西行來到當時的 JavaSoft，因爲我很清楚，那兒是我可以發揮的地方。在這五年內，我擔任 Java 平台程式庫（標準程式庫）的結構設計師。我設計並實現（以及維護）許多程式庫，同時也擔任其他許多程式庫的顧問。隨著 Java 的日益成熟，主持這些程式庫成爲一個人一生中難得的機會。不誇張地說，我有幸和當代許多最傑出的軟體工程師一同工作。在此期間，我學到了有關 Java 語言的許多東西 — 它能做什麼，不能做什麼，如何最有效地運用這種語言及其程式庫。

我企圖透過本書和你分享我的經驗，使你在獲得和我一樣的成功的同時，避免我曾經犯過的錯誤。我借用 Scott Meyers 的《*Effective C++*》[Meyers98] 書籍格式，該書共有 50 個條款，每個條款提出一條改善程式效能和設計的具體準則。我覺得這種格式非常有效，因此我也選擇這麼做。

在許多例子中，我獲得授權，得以引用 Java 程式庫的真實作法來說明相應的條款。描述某些不很完美的工作時，我儘量使用自己寫的碼，但偶而也會引用其他同事的碼。儘管我竭力不使這種局面發生，但如果真的冒犯了人，我在這裡致以誠摯的歉意。提出負面例子，並無責備之意，而是出於協同作用（cooperation）的考量，以便讀者能夠從我們的錯誤經歷中得到啓示。

儘管本書並非只針對「可復用組件」（reusable components）開發人員而撰寫，但過去 20 年來我在這方面的工作肯定無可避免地影響到這本書。我會自然而然地以 exported API 的思路來思考問題，而且我也鼓勵你這麼做。儘管你不一定要開發可

復用組件，但是以這種方式思考，有助於提高軟體設計品質。在認識不足的情況下動手撰寫「可復用組件」的例子並不罕見：你寫了一些有用的東西，在伙伴之間分享，不久後有了許多用戶，這時不能再隨心所欲地修改 API 了。這時候，如果你當初撰寫這個軟體時曾經於 API 設計上付出過心力，你會額手稱慶。

我把焦點放在 API 的設計上，這對於那些醉心於「新式的輕型軟體開發方法學」（new lightweight software development methodology）一例如 **Extreme Programming**（XP, 極限編程）[Beck99] — 的讀者也許有點不自在。這一類方法學強調寫出最簡單的程式，只要能夠運作就好。其實，如果你正採用這一類方法學，你會發現，花些心力在 API 的設計上，對於重構（**refactoring**）過程很有好處。重構的基本目標是改善系統結構和避免程式碼重複冗贅。如果系統組件缺乏設計良好的 API，那麼這些目標是不可能達成的。

世上沒有完美的語言，但有些語言很傑出。我發現 Java 語言和其程式庫，對於程式的品質和效率的提昇非常有幫助。使用 Java，是一種工作上的享受。我希望這本匯聚我的熱情和希望的作品，可以使你更高效、更輕鬆地運用 Java！

Joshua Bloch
Cupertino, California
April, 2001

致謝

Acknowledgment

我要感謝 Patrick Chan 建議我寫這本書，並將這樣的想法傳達給 Lisa Friendly, 系列叢書主編；Tim Lindholm, 系列叢書技術編輯；以及 Mike Hendrickson, Addison-Wesley Professional 的執行編輯。感謝 Lisa、Tim，和 Mike 對我的鼓勵，以及認定我終有一天可以完成此書而抱持的耐心和信心。

感謝 James Gosling 和其原始團隊給予我這麼美好的寫作題材。也感謝眾多追循 James 足跡的 Java 平台工程師。特別我要感謝我在 Sun 公司的 Java Platform Tools 和 Libraries Group 的同事們，感謝他們的理解，他們的鼓勵，他們的支持。後一個團隊由 Andrew Bennett, Joe Darcy, Neal Gafter, Iris Garcia, Konstantin Kladko, Ian Little, Mike McCloskey 和 Mark Reinhold 組成。前一個團隊的成員包括 Zhenghua Li, Bill Maddox 和 Naveen Sanjeeva。

我要謝謝我的經理 Andrew Bennett 和我的協理 Larry Abrahams 對於這個寫作計劃的熱情支持。我要謝謝 Java Software 工程副總裁 Rich Green，他提供的環境讓工程師得以創造性的方式自由思考並發表成果。

我擁有一個你所能夠想像的最佳復閱小組，我要把我最真誠的感謝獻給他們：Andrew Bennett, Cindy Bloch, Dan Bloch, Beth Bottos, Joe Bowbeer, Gilad Bracha, Mary Campione, Joe Darcy, David Eckhardt, Joe Fialli, Lisa Friendly, James Gosling, Peter Haggar, Brian Kernighan, Konstantin Kladko, Doug Lea, Zhenghua Li, Tim Lindholm, Mike McCloskey, Tim Peierls, Mark Reinhold, Ken Russell, Bill Shannon, Peter Stout, Phil Wadler，以及兩位未具名的復閱者。他們提出很多建議，使本書獲

得極大改善，使我得以避免許多難堪。任何剩下來的難堪都是我個人的責任。

許多同行，涵括 Sun 公司內外，參與了本書的技術討論，改善了本書品質。其中 Ben Gomes, Steffen Grarup, Peter Kessler, Richard Roda, John Rose 和 David Stoutamire 貢獻了極有用的深刻見解。我還要特別感謝 Doug Lea，他像是本書許多構想的一塊共鳴板（sounding board，意見試探對象）。Doug 將其時間和知識無窮盡地大方分享給我。

感謝 Julie Dinicola, Jacqui Doucette, Mike Hendrickson, Heather Olszyk, Tracy Russ，以及 Addison-Wesley 的整個支援團隊，謝謝你們的支持和專業。即使處於一種幾乎不可能的緊繃進度中，你們還是保持友善和通融。

謝謝 Guy Steele 爲我寫序。我很榮幸他的參與。

最後，我要謝謝我的妻子，Cindy Bloch，謝謝她的鼓勵和偶而催促我撰寫本書，謝謝她閱讀本書每一條剛出爐的條款；謝謝她操作 Framemaker 幫助我，謝謝她編寫本書索引，也謝謝她在我寫作期間提供我溫暖的生活。

1

緒論

Introduction

本書的目標是幫助你最有效地運用 Java 編程語言及其基本程式庫：`java.lang`, `java.util`，和一部分 `java.io`。本書偶而也會討論其他程式庫，但不涵蓋 GUI（圖形用戶界面）編程或企業級 APIs。

本書由 57 個條款組成，每一條款傳達一項規則。這些規則反映出最有經驗的程式員在實踐過程中的一些有益作法。這些條款被寬鬆地歸類爲 9 章，每一章關心軟體設計的一個概略方向。本書並不企圖寫成必須逐頁閱讀的方式，每個條款有相當程度的獨立性。這些條款有著份量很重的交叉索引，因此你可以輕易在本書各處標定出自己想學習的課程。

大部份條款都以程式實例詳加解說。本書的一個突出點就是，這些實例解說了許多設計範式（design patterns）和慣用手法（idioms）。某些範式或手法可能你已耳熟能詳，例如 ***Singleton***（**條款 2**），其他對你可能很新穎，例如 ***Finalizer Guardian***（**條款 6**）和 ***Defensive readResolve***（**條款 57**）。書後有一個獨立索引（**p.239**）可讓你輕易定位出這些設計範式和慣用手法。某些適當地點他們還被交叉索引至範式領域的標準參考讀物 [Gamma95]。

許多條款內含一個或多個程式實例，說明某些應該避免的作法。這種有時被稱爲「僞範式」（antipatterns）的作法，將以註釋清楚標示出來：`"// Never do this!"`。條款內會解釋爲什麼這個例子不好，並建議替代作法。

本書並非爲初學者而寫：本書假設你已熟悉 Java 編程語言。如果不是這樣，請考慮先閱讀一本好的 Java 編程入門書籍 [Arnold00, Campione00]。雖然本書的設計是給任何一位已有 Java 語言實務經驗的人閱讀，但它應該也能提供給高階程式員一些好糧食。

本書所提的大部分規則都由一些基本原則導出。「清晰」和「簡單」擁有至高無上的重要性。模組（module）的行爲不應該造成用戶的驚訝（譯註：否則這個模組就是不夠清晰）；模組應該儘可能小但又不能太小。（本書所謂模組（module）意指任何可復用的軟體組件（reusable software component），規模從個別函式到多個 packages 所組成的複雜系統。）程式碼應該被復用，好過被複製。模組間的依存性（dependencies）應該儘可能小。錯誤應該在發生後儘快 — 最好是在編譯期 — 被偵測出來。

雖然本書所列規則並非任何時候百分之百適用，但它們的確在許多方面描繪出最佳編程實踐手法。你不應該盲目遵循這些規則，但只有在你擁有足夠的好理由時才可以違反它們。學習編程技藝，就像大多數其他訓練一樣，首先應該學習規則，然後學習何時可以打破規則。

大部分時候，本書並不談效率，而是談如何撰寫清晰、正確、可用、強固、彈性、易維護的程式碼。如果你可以做到這些，通常也就可以輕鬆獲得你需要的效率（**條款 37**）。某些條款的確討論了效率，其中一些甚至提供了實測數字。這些數字通常會伴隨「在我的機器上」這樣的話，你應該把它們視爲近似值。

必要一提的是，我的機器是一部老舊的家用電腦，400 MHz Pentium II，128M RAM，在 Microsoft Windows NT 4.0 上執行 Sun 1.3 Java 2 Standard Edition Software Development Kit（SDK）。這個 SDK 包括 Sun Java HotSpot Client VM，這是一個最先進的 JVM 實作產品，設計用來供客戶端（client）使用。

討論 Java 語言和其程式庫特性時，有時候我們需要明確指出哪一個版本。爲了簡潔起見，本書使用工程版本號碼，而非正式發行名稱。**表格 1.1** 顯示工程版本號碼與正式發行名稱之間的對應關係。

表格 1.1：Java 平台版本

正式發行名稱 Official Release Name	工程版本號碼 Engineering Version Number
JDK 1.1.x / JRE 1.1.x	1.1
Java 2 Platform, Standard Edition, v 1.2	1.2
Java 2 Platform, Standard Edition, v 1.3	1.3
Java 2 Platform, Standard Edition, v 1.4	1.4

雖然某些條款討論了 1.4 版的特性，但除了少數例外，大多數程式實例並不運用那些特性。全部實例已在 1.3 版通過編譯。大部分的它們（說不定是全部）應該可以不加任何修改就在 1.2 版執行。

儘管這些例子都十分完整，不過它們更重視可讀性而非完整性。它們自由運用 packages `java.util` 和 `java.io` 中的各個 classes。爲編譯這些實例，你或許必須自行加上一或兩行「匯入述句」（import statements）：

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

其他照本宣科一成不變的東西也像這樣地被我刻意遺漏。本書支援網站 http://java.sun.com/docs/books/effective 內含每個例子的完整版本，你可以編譯它並執行它。

很大程度上本書使用的術語是《*The Java Language Specification, Second Edition*》[JLS] 中定義的術語。其中某些術語值得特別提出來。Java 語言支援四種型別（types）：**interfaces**（介面）, **classes**（類別）, **arrays**（陣列）, 和 **primitives**（基本型別）。前三種是所謂的 *reference types*。class 所產生的實體及 arrays 都是物件（objects），基本型數值（*primitive values*）則不是。class 係由所謂的欄位（fields）、函式（methods）、成員類別（member classes）以及成員介面（member interfaces）組成。所謂函式的署名（method signature）係由函式名稱和其形式參數（formal parameters）的型別組成，不包括函式回返值型別（return type）。

本書使用了一些不同於《*The Java Language Specification, Second Edition*》的術語。和該書不一樣的是，本書把繼承（**inheritance**）當做 **subclassing** 的同義詞。本書不再使用介面繼承（inheritance for interfaces）這個說法，而僅僅說「某個 class 實現了（*implements*）某個 interface」，或說「某個 interface 擴展了（*extends*）另一個 interface」。爲描述「無明確指定」的存取級別（access level），本書採用的術語是 **package-private**，而非比較技術正確的術語 default access [JLS, 6.6.1]。

本書還用了一些並未定義於《*The Java Language Specification, Second Edition*》的術語。當我說 **exported API** 或說 **simply API**，我的意思是泛指程式員「可藉以存取一個 class, interface 或 package」的所有 classes（類別）, interfaces（介面）, constructors（建構式）, members（成員）和 serialized forms（序列化格式）。術語 **API** 是 application programming interface 的縮寫，這裡使用它而不使用另一個也被大眾喜愛的術語 interface，是爲了避免和 Java 語言構件中的 `interface` 混淆。程式員撰寫程式時

如果使用了某個 API，我們就說他是那個 API 的用戶（**user**）。如果 class 的實作碼中運用了某個 API，我們稱它是那個 API 的客戶（**client**）。

classes（類別）, interfaces（介面）, constructors（建構式）, members（成員）和 serialized forms（序列化格式）統稱爲「**API 元素**」。所謂 **exported API** 就是由 API 元素組成，可在其定義所在之 package 外部被存取。任何客戶都可以使用這些 API 元素，而 API 的作者負責提供支援。Javadoc 工具程式在其預設操作模式中產生的文件，也正是和這些元素有關。我們可以寬鬆地說，一個 package 所含的 exported API 由以下東西組成：package 內的每一個 public class 或 interface 所擁有的 public 和 protected 成員和建構式（constructors）。

2

創建和銷毀物件

Creating and Destroying Objects

本章討論物件的創建（creating）和銷毀（destroying）：何時以及如何創建物件，何時以及如何避免創建物件，如何確保被銷毀的物件處於適當時機之下，如何管理任何必須於物件銷毀之前完成的清理動作（cleanup actions）。

條款 1：考慮以 "static factory methods" 取代建構式

一個 class 如果允許客戶獲得它的一個實體，正常的方法是提供一個 public 建構式。但是有另一個較少爲人知的技術，也應該成爲每一位程式員工具箱的一部分。class 可以提供一個所謂的 public static ***factory method***，那其實就是一個會傳回 class 實體的 static 函式。下面是一個取自 class `Boolean`（基本型別 `boolean` 之外覆類別）的簡單例子，其 static factory method 自 1.4 版加入，將一個 `boolean` primitive value 轉換爲一個 `Boolean` object reference：

```
public static Boolean valueOf(boolean b) {
    return (b ? Boolean.TRUE : Boolean.FALSE);
}
```

class 可以在建構式之外額外添加 static ***factory method***s，或是拿它來取代建構式。以 static ***factory method*** 取代 public 建構式，有兩個優點和兩個缺點。

優點之一是，它和建構式不同，它有自己的名稱。如果建構式的參數之中沒有一個能夠清楚描述被傳回的物件，那麼改而採用帶有良好名稱的 static ***factory***，可以使得 class 較易被使用，並使程式碼較易被閱讀。例如建構式 `BigInteger(int, int, Random)`，會傳回一個或許是質數的 `BigInteger`，如果替換爲一個名爲 `BigInteger.probablePrime` 的 static ***factory method***，似乎解釋效果較佳。（這個 static ***factory method*** 終於在 1.4 版被加入了。）

class 的每一個建構式都必須有不同的署名（signature）。程式員已經知道如何迴避這一限制：提供兩個建構式並令其參數列的惟一不同在於其參數的次序不同。這不是個好主意。這種 API 的用戶絕對無法記住哪一個建構式是哪一個，最終很可能發生錯誤的呼叫。閱讀這種程式碼的人在未曾參考 class 說明文件的情況下也無法知道程式做了些什麼。

由於 static ***factory method***s 可以擁有自己的函式名稱，所以它們不會遭受「署名式相同的建構式只能有一個」的限制。因此，如果某個 class 需要多個相同署名的建構式，你應該考慮以 static ***factory method***s 取代其中一個或多個建構式，並謹慎爲它們命名，突顯其間的差異。

static ***factory method***s 的第二個優點是，它和建構式不同，不需要每次被呼叫都創建一個新物件。這使得 *immutable classes*（**條款 13**）可使用預先建構好的實體，或是在實體被建構時便以快取（cache）形式儲存起來而後反覆分發（*dispense*）這些實體，以避免創建出非必要的重複物件。`Boolean.valueOf(boolean)`函式可說明這項技術：它從不創建物件。如果等價物件（equivalent objects）索求頻繁，這項技術可以大大改善效率 — 特別是如果這些物件的創建很耗成本的話。

「不同的 static ***factory method***s 可在反覆呼叫中傳回相同物件」的能力，也可以用來嚴厲控制任何已知時間可以存在什麼樣的實體。有兩個理由需要那麼做，第一，這使得 class 得以保證自己是個 ***singleton***（**條款 2**）。第二，這使 *immutable* class 得以確保不會有兩個相等實體(equal instances)存在，也就是說它確保「`a.equals(b)` 若且惟若 `a==b`」。如果 class 有了這樣的保證，那麼它的客戶就可以使用 `==` 運算子取代 `equals(Object)`函式，這可能導致結結實實的效率改善。描述於**條款 21** 的 ***typesafe enum*** 範式實現了這項最佳化措施，`String.intern()`函式也在一個有限形式中實現了它。

static ***factory method***s 的第三個優點是，它和建構式不同，它可以傳回一個「隸屬於函式回返值型別之任何子型別（subtype）」的物件。這使你在「選擇回返物件之型別」這件事情上擁有很大彈性。

這種彈性的應用之一就是：API 可以傳回 non-public classes 的一個物件。以此種方式來隱藏 implementation classes，可以獲得一個簡潔的（*compact*）API。這項技術很適合用於 interface-based frameworks，在其中，interfaces 爲 static ***factory method***s 提供了自然回返型別（natural return types）。

舉個例子，Collections Framework 針對其 collection interfaces 有將近 20 個很便利的實作類別，提供不可改動的（unmodifiable）、同步的（synchronized）或其他性質的 collections。這些實作類別主要便是透過一個不可實體化（noninstantiable）的 class（`java.util.Collections`）中的 static ***factory method***s 匯出（*exported*）。傳回的物件全都隸屬於 nonpublic classes。

目前的（上述的）collections Framework API 規模遠遠小於「針對 20 個各自不同的 public classes 進行匯出」所需的規模。目前作法的好處是，不僅 API 數量（體積）減少了，概念上的重量也減少了。用戶知道被傳回的物件嚴格擁有相關之 interface 所明確指定的 API，所以不需閱讀額外的 class 說明文件。此外，使用這樣一個 static ***factory method*** 可使得客戶係透過 interface 而非透過 implementation class 來指涉（引用）被傳回的物件，這是比較理想的實踐作法（**條款 34**）。

public static ***factory method*** 傳回的物件不僅可以是個 nonpublic class 物件，而且視 static ***factory*** 函式參數的不同，還可以涉及不同的 nonpublic class — 只要是函式回返型別的任何一個 subtype 都可以。回傳物件所隸屬的 class，也可以隨著發行版本的不同而不同，用以提高軟體的維護性。

被 static ***factory method*** 傳回的物件，其所隸屬的 class 並不需要在「擁有 static ***factory method*** 的那個 class」被撰寫之際就已存在。這樣的彈性構成了 **service provider frameworks** — 例如 Java Cryptography Extension（JCE）— 的基礎。所謂 service provider framework 是一個系統，其中的 providers 會爲 API 製造出多個實作品，使它們可被 framework 的用戶使用。會有一個機制被提供出來負責註冊（登錄）這些實作品，讓它們處於可用狀態。當這種 framework 的客戶使用 API 時，不需要操心它們用的是哪一個實作品。

在 JCE 中，系統管理員（system administrator）註冊一個 *implementation* class 的作法是：編輯一個眾所周知的 **Properties** 檔案，添加一筆條目，將一個 string key（字串鍵值）映射到對應的 class 名稱。客戶使用一個 static ***factory method*** 並接受 key 爲參數。這個 static ***factory method*** 在一個 map 中尋找 `Class` 物件，該 map 根據

Properties 檔初始化，並以 `Class.newInstance()`函式對這個 class 進行實體化。下面的實作梗概說明了此一技術：

```
// Provider framework sketch
public abstract class Foo {
    // Maps String key to corresponding Class object
    private static Map implementations = null;

    // Initializes implementations map the first time it's called
    private static synchronized void initMapIfNecessary() {
        if (implementations == null) {
            implementations = new HashMap();

            // Load implementation class names and keys from
            // Properties file, translate names into Class
            // objects using Class.forName and store mappings.
            ...
        }
    }

    public static Foo getInstance(String key) {
        initMapIfNecessary();
        Class c = (Class) implementations.get(key);
        if (c == null)
            return new DefaultFoo();

        try {
            return (Foo) c.newInstance();
        } catch (Exception e) {
            return new DefaultFoo();
        }
    }
}
```

static ***factory method***s 的主要缺點是，classes 如果沒有了 public 或 protected 建構式（譯註：被 ***factory method***s 取代），將無法被 *subclassed*。這一點對於被 public static ***factories*** 傳回的 nonpublic classes 也一樣。例如我們不可能 *subclassing* Collections Framework 中的任何 *implementation* classes。有人把這視爲一種淨化，因爲這可以鼓勵程式員以複合（composition）取代繼承（inheritance）（**條款 14**）。

static ***factory method***s 的第二個缺點是，它們無法明顯地和其他 static 函式有所區分。它們在 API 文件中不像建構式那般引人注目。此外，static ***factory method***s 表現出正常軌道外的一種偏航，因此可能不易從 class 文件中理解如何實體化一個以 static ***factory method***s 取代建構式的 class。這個缺點可以因爲堅守標準命名習慣

而降低 — 這些命名習慣仍在演化之中，但有兩個通俗常見的 static ***factory methods*** 名稱：

- `valueOf` — 傳回一個實體，它寬鬆地說，和其參數擁有相同的值。擁有這種名稱的 static ***factory methods*** 都是一種型別轉換運算子（type-conversion operators）。
- `getInstance` — 傳回一個由其參數描述的實體，但我們不能說它擁有與參數相同的值。以 ***singletons*** 爲例，它傳回惟一一份實體。這個名稱常被 provider frameworks 使用。

總的來說，static ***factory methods*** 和 public 建構式都有它們自己的用途，了解它們彼此的優勢是值得的。請避免在沒有首先考慮 static ***factories*** 的情況下本能反應地選擇建構式，因爲 static ***factories*** 往往比較適用。但如果你權衡這兩種作法而沒有哪一方擁有明顯優勢的話，或許最好還是提供一個建構式，因爲畢竟它是語言規範。

條款 2：以 private 建構式厲行 *singleton*（單件）性質

所謂 ***singleton*** 是指「只能被實體化（instantiated）一次」的 class [Gamma95, p127]。***Singletons*** 通常用來表現某些本質上具有惟一性的系統組件，例如視訊螢幕或檔案系統。

兩種辦法可以實現 ***singletons***，共通點是令建構式爲 private，並提供一個 public static 成員，允許客戶存取那惟一一個實體。作法之一是令 public static 成員成爲一個 final 欄位：

```
// Singleton with final field
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();

    private Elvis() {
        ...
    }

    ... // Remainder omitted
}
```

private 建構式只被呼叫一次，用以初始化那個 public static final 欄位 `Elvis.INSTANCE`。public 或 protected 建構式的缺席可以保證一個「雌雄同體的世界」（譯註：*monoelvistic*，意指惟一無二的）：當 `Elvis` class 被初始化，恰恰就只有一個 `Elvis` 實體存在，不多也不少。客戶無法改變這一事實。

第二種作法是，提供一個 public static ***factory method***，取代上述的 public static final 欄位：

```
// Singleton with static factory
public class Elvis {
    private static final Elvis INSTANCE = new Elvis();

    private Elvis() {
        ...
    }

    public static Elvis getInstance() {
        return INSTANCE;
    }

    ... // Remainder omitted
}
```

對 static 函式 `Elvis.getInstance()`的任何呼叫動作，都獲得同一個 object reference，不會有任何其他 `Elvis` 實體被創建出來。

第一種作法的主要優點是，成員的宣告式中包含了 class，這使它得以比較清楚地表示這個 class 是個 ***singleton***：由於那個 public static 欄位是個 final，所以該欄位將總是內含相同的 object reference。這一事實也可能爲此作法帶來輕微的效率優勢，不過優秀的 JVM 實作品應該能夠在以下的第二種作法中將 static ***factory method*** 的呼叫動作內聯化（inlining）而消除這項效率優勢。

第二種作法的主要優點是給你彈性，讓你得以改變心意，使得在不必改變 API 的情況下令這個 class 不再成爲一個 ***singleton***。一個針對 ***singleton*** 而設計的 static ***factory method*** 應該傳回 class 的惟一實體，但亦可輕易修改，改傳回（例如）一個「對喚起此函式之任何執行緒而言」獨一無二的實體。

總的來說，如果你絕對確定某個 class 永遠保持爲一個 ***singleton***，那麼對它使用第一種作法是很合理的。如果你希望保留一點餘地，請使用第二種作法。

如果要讓一個 ***singleton*** class 成爲 serializable（第 10 章），只在其宣告式中加上 "`implements Serializable`" 是不夠的。爲了維持 ***singleton*** 的保證，你還必須提供一個 `readResolve()`（**條款 57**）。否則一個序列化（*serialized*）實體經過反序列化（*deserialization*）後，會導致創建出一個新實體，在我們的例子中也就導致僞造出一個 `Elvis`。欲阻止這樣的事情，請爲 `Elvis` class 加上如下的 `readResolve()`：

```
// readResolve method to preserve singleton property
private Object readResolve() throws ObjectStreamException {
    /*
     * Return the one true Elvis and let the garbage collector
     * take care of the Elvis impersonator.
     */
    return INSTANCE;
}
```

這個條款和**條款 21**（描繪 ***typesafe enum*** 範式）其實反映出相同的主題。在這兩種情況下，private 建構式被用來和 public static 成員產生關係，以確保 class 在初始化之後不會有新的實體再被創建出來。本條款中，只有一個實體會被創建；**條款 21** 之中會爲 enumerated type（列舉型別）的每一個成員創建出一個實體。在下一個條款（**條款 3**）中，這種作法有更進一步的發揮：不存在任何 public 建構式，以便確保 class 沒有任何實體被創建出來。

條款 3：以 `private` 建構式厲行不可實體化性質（noninstantiability）

偶爾你會想寫個 class，它只由一群 static 函式和 static 欄位組成。如此的 classes 名聲不太好，因爲有些人誤用它們在物件導向語言中撰寫程序式（procedural）程式。儘管如此，它們確實也有自己的用途。我們可以用它來聚集作用於 primitive values 或 arrays（例如 `java.lang.Math` 或 `java.util.Arrays`）上的函式，或是用來聚集作用於「實現某一特殊介面（例如 `java.util.Collections`）」之物件身上的 static 函式。也可以用它來聚集作用於 final class 身上的函式，以替代對 class 的擴展（extending）行爲。

如此的 utility classes 並不打算被實體化（instantiated）。如果它有一個實體，反倒有點荒謬。由於它未曾明確定義任何建構式，編譯器會供應一個 public 且無參數的 *default* 建構式給它。對用戶而言，這個建構式和任何其他建構式並無不同。在許多 APIs 中我們可以看到一些無意造成的「可實體化 class」，這種情況並不罕見。

企圖「將 class 變成 abstract class」藉以實現「不可實體化」其實是沒有用的。這樣的 class 可以被 *subclassed*，而獲得的 subclass 可以被實體化。此外這種作法會誤導用戶以爲這樣的 class 被設計用來繼承（**條款 15**）。有一個簡單手法可以確保「不可實體化」性質。要知道，當 class 沒有內含任何明確定義的建構式時，編譯器才會自動爲它生出一個 *default* 建構式，因此任何 class 只要內含一個明確定義的 private 建構式，而且不再有其他建構式，就可以使自己「不可實體化」了：

```
// Noninstantiable utility class
public class UtilityClass {

    // Suppress default constructor for noninstantiability
    private UtilityClass() {
        // This constructor will never be invoked
    }
    ...  // Remainder omitted
}
```

由於明確定義的建構式是個 private，它在 class 範圍之外是不能被取用的，因而保證這個 class 絕不會被實體化 — 如果這個建構式不被 class 本身喚起的話。這種手法有點反直覺，因爲建構式的刻意用途竟然是保證它無法被喚起。因此最好爲它加上一段註釋，描述此種建構式的撰寫目的。

連帶影響的是，這種手法也阻止了 class 被 *subclassed*。因爲所有建構式都必須喚起一個可用的 superclass 建構式 — 不論是經過明確定義的，或是隱晦的（譯註：指 *default* 建構式），然而在此手法之下，subclass 無法喚起任何可用之建構式。

條款 4：避免創建「重複物件」（duplicate objects）

通常，重複使用（復用, reuse）同一個物件，比起每次需要時創建一個嶄新而功能相同的物件，要合適些。復用（reuse）不但比較快，也比較流行。物件如果是不可變的（*immutable*, **條款 13**）就總是可以被復用。

做爲一個極端反例，讓我們考慮這樣一個述句：

```
String s = new String("silly");  // DON'T DO THIS!
```

此句每次執行便創建一個新的 `String` 實體，然而沒有一個是必要的。`String` 建構式的引數 `"silly"` 本身就是個 `String` 實體，其功能完全等於被該建構式創建出來的任何物件。如果這種用法發生於一個迴圈或在一個頻繁被喚起的函式中，說不定會創建出數百萬個非必要的 `String` 實體。

改良版本如下：

```
String s = "No longer silly";
```

這個版本只使用單一 `String` 實體，而不是每次執行時創建一個新實體。此外它也保證這個物件可被執行於相同虛擬機器上、內含相同字串字面常數（string literal）的任何其他程式復用 [JLS, 3.10.5]。

通常，如果 *immutable* classes 同時提供 static ***factory method***s（**條款 1**）和建構式，你可以採用 static ***factory method***s 以避免創建重複物件。例如 static ***factory method*** `Boolean.valueOf(String)` 就幾乎總是比其建構式 `Boolean(String)` 更受歡迎。建構式每次被呼叫便會創建出一個新物件，但 static ***factory method*** 不一定。

除了復用 *immutable* 物件，你也可以復用那些你知道它們其實不會被更改的 *mutable* 物件。下面是一個稍帶微妙但十分普遍的例子，告訴你什麼事不要做，涉及的是「數值一旦計算出來就絕不會再被更改」的一些 *mutable* 物件：

```
public class Person {
    private final Date birthDate;
    // Other fields omitted

    public Person(Date birthDate) {
        this.birthDate = birthDate;
    }
```

```
    // DON'T DO THIS!
    public boolean isBabyBoomer() {
        Calendar gmtCal =
            Calendar.getInstance(TimeZone.getTimeZone("GMT"));
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
        Date boomStart = gmtCal.getTime();
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
        Date boomEnd = gmtCal.getTime();
        return birthDate.compareTo(boomStart) >= 0 &&
               birthDate.compareTo(boomEnd)   <  0;
    }
}
```

上述的 `isBabyBoomer()`每次被喚起便非必要地創建了一個嶄新的 `Calendar` 實體，一個 `TimeZone` 實體，和兩個 `Date` 實體。下面的版本避免了這種低效率的動作，代之以一個 static initializer：

```
class Person {
    private final Date birthDate;

    public Person(Date birthDate) {
        this.birthDate = birthDate;
    }

    /**
     * The starting and ending dates of the baby boom.
     */
    private static final Date BOOM_START;
    private static final Date BOOM_END;

    static {
        Calendar gmtCal =
            Calendar.getInstance(TimeZone.getTimeZone("GMT"));
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
        BOOM_START = gmtCal.getTime();
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
        BOOM_END = gmtCal.getTime();
    }

    public boolean isBabyBoomer() {
        return birthDate.compareTo(BOOM_START) >= 0 &&
               birthDate.compareTo(BOOM_END)   <  0;
    }
}
```

改善後的 `Person` class 只有在它被初始化的時候才惟一一次創建 `Calendar`, `TimeZone`, 和 `Date` 實體，而不再是每次 `isBabyBoomer()`被喚起就創建它們。如果該函式被頻繁呼叫的話，這便可導致重大的效率提昇。在我的機器上，原先版本進行一百萬次呼叫花費 36,000 ms，改善版本只花了 370 ms，幾乎有 100 倍快。不只是效率提昇，程式碼也比較清晰。將 `boomStart` 和 `boomEnd` 從區域變數改爲 final static 欄位，可使讀者更清楚知道這些日期被視爲常數，程式碼較易被理解。不過這種最佳化所帶來的效率變化並非總是如此戲劇性，此處是因爲 `Calendar` 實體的創建成本很高。

如果 `isBabyBoomer()`絕不會被喚起，上述 `Person` class 改善版本將會非必要地初始化 `BOOM_START` 和 `BOOM_END` 欄位。如果我們在 `isBabyBoomer()`第一次被喚起時對這些欄位實施 lazily initializing（**條款 48**），就有可能消除這種非必要的初始化動作，但此法並不是很受推薦。就像 lazy initialization 常常帶來的情況一樣，這會使實作手法更趨複雜，不太可能導致顯著的效率提昇（**條款 37**）。

本條款先前所有例子中，很明顯我們所討論的物件可以被復用，因爲它們被初始化之後就沒有再被改動過。但是另有一些情況不是那麼明顯。考慮 ***adapter***s（配接器） [Gamma95, p139]，又稱爲 ***view***s（映件）。所謂 ***adapter*** 是這樣一個物件：將任務委託（delegates）給一個背景物件（backing object），爲背景物件提供另一個（正常之外的）介面。由於 ***adapter*** 並無任何狀態（state）超越其背景物件，所以我們沒有必要爲一個已知物件的一個已知 ***adapter*** 創建一個以上的實體。

舉個例子，`Map` interface 的 `keySet()`傳回該 `Map` 物件的一個 `Set` view，由 map 中的所有鍵值（keys）組成。單純地想，似乎每次呼叫 `keySet()`都必須創建出一個新的 `Set` 實體，但事實上每次對同一個 `Map` 物件呼叫 `keySet()`，傳回的卻是同一個 `Set` 實體。雖然被傳回的 `Set` 實體往往是可變的（*mutable*），但所有這些被傳回物件在機能上完全一致：當某個被傳回物件有所改動，其他每一個被傳回物件也都發生變化，因爲它們的背後是同一個 `Map` 實體。

請不要誤以爲本條款暗示物件的創建非常昂貴所以應該避免。事實上，那種「建構式沒做什麼動作」的小型物件，其創建和回收都十分廉價，尤其是在現代（高強的）JVM 實作品中。因此，產生額外物件（譯註：而非如本條款所強調的復用單一物件）以提高程式的清晰度、單純性和威力，往往也是很棒的一件事情。

爲了避免創建物件，因而自行維護自己的物件池（object pool），是個糟糕的主意，除非池中物都是重量級物件（譯註：意指創建成本高昂）。一個適合使用物件池的典型例子就是資料庫連線（database connection）。建立這樣一個連線的成本很高，因此復用這些物件就很合理。一般而言，維護自己的物件池會使你的程式碼變得雜亂，增加記憶體用量，並且對效率有所蘄傷。現代化 JVM 實作品都有高度最佳化的垃圾回收器（garbage collectors），可輕易勝過輕量級物件的物件池。

與本條款相互輝映的是**條款 24**，其中談論「保護性拷貝」（defensive copying）。本條款說：如果你應該復用一個既有物件，就不該創建新物件。**條款 24** 卻說：如果你應該創建一個新物件，就不該復用既有物件。注意，如果我們在應該使用「保護性拷貝」的場合卻採行「物件復用」手段，付出的代價遠遠超過非必要地創建重複物件。是的，在需要實施「保護性拷貝」的場合中卻沒有成功地那麼做，會導致潛在錯誤和安全漏洞（security holes）；而創建非必要物件卻只不過是影響程式的風格和效率而已。

條款 5：消除老舊的（逾期的）object references

當你從一個必須手動進行記憶體管理的語言如 C 或 C++，轉換到一個自動進行垃圾回收（garbage-collected）的語言時，做爲一個程式員，你的工作輕鬆多了，因爲物件會在被用過之後自動歸還給系統。初次面對這種機制時，你會感覺那像是一個魔術。這種技術很容易給你一個印象：你再也不需要操心記憶體管理這檔事了。但這並不是真的。

考慮下面這一份簡單的 stack 實作碼：

```
// Can you spot the "memory leak"?
public class Stack {
    private Object[] elements;
    private int size = 0;

    public Stack(int initialCapacity) {
        this.elements = new Object[initialCapacity];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        return elements[--size];
    }

    /**
     * Ensure space for at least one more element, roughly
     * doubling the capacity each time the array needs to grow.
     */
    private void ensureCapacity() {
        if (elements.length == size) {
            Object[] oldElements = elements;
            elements = new Object[2 * elements.length + 1];
            System.arraycopy(oldElements, 0, elements, 0, size);
        }
    }
}
```

其中並沒有明顯錯誤。無論你如何測試，它都可以通過考驗。但其中存在一個潛在問題。寬鬆地說，這個程式存在記憶體洩漏（memory leak）問題，可能會默默

顯露在由於「增加垃圾回收器的活動」或由於「增加記憶體用量」而導致的效率降低上。極端情況下，這般記憶體洩漏可能造成頻繁的磁碟分頁（*paging*）動作，甚至因爲 `OutOfMemoryError` 而造成程式停擺，不過這種情況比較罕見。

那麼，「記憶體洩漏」發生在哪兒呢？是的，如果這個 stack 成長而後縮小，被彈出（*popped off*）的那些物件並不會被回收 — 縱使使用這個 stack 的程式不再引用它們。這是因爲 stack 仍然維護著指向那些「不再被使用的物件」的老舊（逾期）的 references，。所謂老舊（逾期）的 reference，意指一個再也不會被提領（*dereferenced*）的 reference。本例位於 `element` array 之中「有作用的部位」（active portion）外的任何 references 都是老舊（逾期）的。所謂「有作用的部位」（active portion）」係由「索引值小於 array `size`」的元素組成。

在支援垃圾回收機制的語言中，「記憶體洩漏」（更適當的說法是「無意識的物件保留」）是隱伏、潛沉而隱蔽的。如果一個 object reference 被非刻意地留住，那麼不只那個物件不會受到垃圾回收機制的眷顧，該物件所引用（指涉）的任何其他物件也都如此。縱使只有少數一些 object references 被非刻意地留住，也可能導致許多物件不再被自動回收，進而大大影響效率。

修正這種問題的作法十分簡單：一旦 references 逾期（老舊），就把它設爲 null。 先前的 `Stack` class 中，只要某個元素彈出 stack，其 reference 就老舊（逾期）了。正確的 `pop()`應該是這樣：

```
public Object pop() {
    if (size==0)
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null; // Eliminate obsolete reference
    return result;
}
```

將逾期的 references 設爲 null 的另一個好處是，如果它們後來又被（不正確地）提領（*dereferenced*），程式立刻會因爲 `NullPointerException` 而失敗，這可比安安靜靜地做些錯事來得好，有助於我們儘快偵測出編程錯誤。

當程式員初次被這種問題螫痛後，他們會有過度補償的傾向：在完成對 object reference 的運用之後立刻將它們都設爲 null。這既非必要也不是我們所期望的，因爲它會使程式碼變得雜亂，並且可想而知會降低效率。將 object reference 設爲 null 應該是一種例外而不是一種常規。老舊 reference 的最佳消除辦法是復用它所含的

那個變數，或是讓它掉離作用域（scope）— 如果你將每個變數定義於儘可能窄的作用域中，這種情況便會很自然地發生（**條款 29**）。值得注意的是，在目前的 JVM 實作品中，僅僅離開變數定義式所在的程式碼區塊是不夠的，必須離開其所在的函式，才能讓 reference 徹底消失（譯註：被回收）。

那麼，何時應該將一個 reference 設爲 null 呢？`Stack` class 的哪一方面使它遭受記憶體洩漏的懲罰？簡而言之，問題在於 `Stack` class 管理了它自己的記憶體。儲存池（storage pool）由 `elements` array 的元素組成（它們都是 object reference，而不是 objects 本身）。array 作用部位（active portion）內的元素都已獲得配置，而作用部位以外的元素尚未配置。垃圾回收機制無法知道這一點。對垃圾回收器而言，`elements` array 內的所有 object references 都同樣有效。只有程式員才知道 array 的非作用區（inactive portion）是不重要的。因此我們在 array 元素一旦變成非作用區的一部分時，就令該元素爲 null，這就有效地將此事通知了垃圾回收器。

一般而言，任何時候只要 class 管理了它自己的記憶體，程式員就應該對記憶體洩漏保持警惕。此例之中，任何時候只要某個 array 元素被釋放，該元素所含的 object references 就應該被設爲 null。

記憶體洩漏的另一個常見原因是 caches（快取裝置）。當你將一個 object reference 放進 cache 之中，很容易就會忘記它在那兒，並因而在它不再有用之後，仍然把它遺留於 cache 之中。這個問題有兩個解法。如果很幸運地你所實現的是這樣的 cache：「只要 cache 之外存在有任何 references 指涉 cache 的某筆條目（entry）的鍵值（key），該筆條目就是有意義的」，那麼請把 cache 實作爲一個 `WeakHashMap`；於是任何條目會在逾期之後被自動移除。然而更常見的情況是，「cache 所存條目是否仍有意義」並無明確定義，只知它會隨著時間逝去而逐漸失去價值。在此情況下，cache 本身應該三不五時地清理廢棄條目。清潔工作可由背景執行緒進行（或許透過 `java.util.Timer` API），或是在「爲 cache 添加新條目」時連帶進行。Java 1.4 添加的 `java.util.LinkedHashMap` class 及其 `removeEldestEntry()`使後一種作法變得更容易。

記憶體洩漏（memory leak）並不像其他明顯錯誤那般地容易顯露出來，所以可能存在於系統多年。通常只有在小心翼翼地程式碼檢閱或在某種除錯工具（所謂 heap 評測器）協助之下，記憶體洩漏問題才會被挖掘出來。因此非常值得在它們發生之前先學習如何對這樣的問題預做準備，並學習如何阻止它們發生。

條款 6：避免使用 finalizers（終結式）

Finalizers（終結式）不可預期，因此往往帶有危險，而且往往並非必要。它們的運用可能造成錯誤的行爲，糟糕的效率，以及移植上的問題。當然 finalizers 也有其用途，稍後我會在本條款中介紹，但是根據經驗，最好避免使用 finalizers。

C++ 程式員不應該把 finalizers 類比爲 C++ 的解構式（destructors）。在 C++ 中，解構式是「歸還物件相關資源」的正當途徑，是建構式的一個必要對比函式。在 Java 語言中，當物件變得不可觸及（unreachable），垃圾回收器便歸還物件佔用的記憶體，過程中不需程式員操任何心。C++ 解構式也被用來歸還其他非記憶體資源，這在 Java 語言中通常以 `try-finally` 區段來達成目的。

沒有任何保證提及 finalizers 會被準時（及時）執行 [JLS, 12.6]。物件變得「不可觸及」之後，直到物件的 finalizer 被執行起來，其間可能相隔任意長度的時間。這意味你絕不應該在一個 finalizer 中進行任何與時間有絕對關聯（time-critical）的任務。例如倚賴 finalizer 關閉被開啓的檔案，是一種嚴重錯誤，因爲被開啓檔案的描述器（open file descriptors）是一種有限資源。如果只因爲 JVM 遲遲沒有執行 finalizers 而使許多檔案處於開啓狀態，程式可能會潰敗，因爲它不再能夠開檔。

「及時執行 finalizers」正是垃圾回收演算法的主要功能之一，會隨著不同的 JVM 實作品而有所變化。程式的行爲如果取決於 finalizers 的及時性，可能會變得不穩定。這樣的程式很有可能在你自己的 JVM 上演出完美，卻在你最重要的顧客的 JVM 上讓你出糗。

遲緩的終結行動並不只是一個理論上的問題，它真的會對實際生活帶來影響。爲 class 提供 finalizer，有可能（雖然頗爲罕見）超乎控制地推遲其物件回收時機。我有一位同事，最近對一個長時間運轉的 GUI 程式進行除錯，該程式神秘地因爲一個 `OutOfMemoryError` 而死亡。分析報告指出，在它死亡時刻，程式中有數千個圖形物件在其 finalizer queue 中等待被終結（而後被歸還）。不幸的是 finalizer 執行緒以一個低於應用程式的優先權運行著，所以物件並未在有資格被終結（歸還）時就真的被終結（歸還）。JLS（Java 語言規格書）不保證哪個執行緒執行 finalizers，所以沒有任何可移植方案可以避免這個問題，除非根本不使用 finalizers。

JLS（Java 語言規格書）不僅不保證 finalizers 被及時執行，也不保證它們終將被執行。一個程式在結束前未能執行某些不再可觸及的物件的 finalizers，是完全有可能的。因此你絕對不能倚賴 finalizer 更改關鍵的（重要的）永續狀態（persistent state）。舉個例子，如果你倚賴 finalizer 釋放一份共享資源，例如資料庫上的永久鎖件（persistent lock），有可能對整個分佈式系統帶來「令人難以忍受的等待」。

不要被 `System.gc()`和 `System.runFinalization()`誘惑。它們確實可能提高 finalizers 的執行可能，但仍舊沒有任何保證。惟一保證執行終結行動（finalization）的函式是 `System.runFinalizersOnExit()`和其有害的雙生兄弟 `Runtime.runFinalizersOnExit()`。這兩個函式都有致命瑕疵，已經不再受到支持。

如果你不確定是否應該避免使用 finalizers，這裡還有一個趣聞值得思考：如果一個未被捕捉的異常在 finalization 期間被拋出，該異常將被忽略，而物件的 finalization 亦將結束 [JLS, 12.6]。未被捕捉的異常可能使物件處於一種走樣（脫軌）狀態。如果另一個執行緒企圖使用如此一個走樣物件，任何不確定的行爲都有可能發生。正常情況下一個未被捕捉的異常會結束執行緒並印出一份 stack 追蹤報告，但如果異常發生於 finalizer 執行期間就不會 — 甚至連個警告都沒有。

那麼，對於那種「物件會封裝某些資源（例如檔案或執行緒），而資源必須被歸還」的 class，你應該做些什麼，才能不必爲它撰寫 finalizer 呢？只要提供一個「明確的終結函式」（explicit termination method），並要求客戶一旦不再需要該 class 的任何實體就喚起該函式，即可。這些物件必須持續追蹤自己是否已被終結，作法是讓上述「明確的終結函式」在一個 private 欄位上記錄此物件是否不再有效，並令其他函式都檢查這一欄位 — 如果函式在物件被終結後才被呼叫，就拋出 `IllegalStateException` 異常。

「明確的終結函式」（explicit termination method）的一個典型例子是 `InputStream` 和 `OutputStream` 的 `close()`。另一個例子是 `java.util.Timer` 的 `cancel()`，它會執行必要的狀態改變，以造成相應於一個 `Timer` 實體的執行緒能夠溫和地終止自己。來自 `java.awt` 的例子還包括 `Graphics.dispose()` 和 `Window.dispose()`，實際應用中它們往往因爲悲慘的效率而受到漠視。另一個函式是 `Image.flush()`，它歸還 `Image` 實體的所有資源，但把該實體留在一個仍可使用狀態，必要時重新配置資源。

「明確的終結函式」（explicit termination method）常常用來與 `try-finally` 構件結合，確保及時完成終結行動。只要在 `finally` 子句中呼叫「明確的終結函式」，便可確保它將會被執行 — 即使物件在被使用過程中拋出異常：

```
// try-finally block guarantees execution of termination method
Foo foo = new Foo(...);
try {
    // Do what must be done with foo
    ...
} finally {
    foo.terminate();  // Explicit termination method
}
```

既然如此，finalizers 到底有什麼用？有的，它們有兩個正當用途。如果物件擁有者忘記呼叫你所提供的「明確的終結函式」，finalizers 可以做爲一個安全網。雖然沒有人保證 finalizer 將被及時喚起，但是當客戶沒有呼叫「明確的終結函式」以致無法達到最終一擊時（但願這種情況儘量不要發生），關鍵性資源稍晚被釋放，總比永遠不被釋放來得好。先前拿來做爲「明確的終結函式」實例的三個 classes（`InputStream`, `OutputStream` 和 `Timer`）也都有 finalizers 做爲安全網，在其「明確的終結函式」未被呼叫時起彌補作用。

finalizers 的第二個正當用途與物件的 **native peer**s（原生同等物）有關。所謂 native peer 是「正常物件藉著 native method 而委託（*delegates*）」的一個 native object。由於 native peer 並非標準（正常）物件，所以垃圾回收器不知道它，也無法在其 **normal peer** 被歸還後將它一併歸還。finalizer 是用來執行這一任務的適當承載工具 — 如果 native peer 不持有關鍵資源的話。但如果 native peer 持有的資源必須被及時終止，class 就應該有個先前所說的「明確的終結函式」，它應該做「釋放關鍵資源所必須做的任何事情」。這個終結函式可以是個 native method，抑或可以喚起一個 native method。

注意，finalizer chaining 並不會被自動執行。如果 class（`Object` 除外）有一個 finalizer，而其 subclass 覆寫了它，subclass finalizer 必須手動喚起 superclass finalizer。你應該在某個 `try` 區段中終結 subclass，並在相應的 `finally` 區段中喚起 superclass finalizer。這樣可以確保 superclass finalizer 一定被執行起來，即使 subclass finalization 拋出異常；反之亦然：

```
// Manual finalizer chaining
protected void finalize() throws Throwable {
    try {
        // Finalize subclass state
        ...
    } finally {
        super.finalize();
    }
}
```

如果 subclass 實作者覆寫（overrides）了一個 superclass finalizer，但是忘記（或刻意不）手動呼叫 superclass finalizer，那麼 superclass finalizer 絕不會被喚起。防禦這種不夠謹慎或帶有惡意的 subclass 的作法是，爲每一個被終結物件創建一個額外物件。當然啦，必須承擔成本。這時我們不再把 finalizer 放在需要終結處理的 class 內，而是把它放進一個 anonymous class（匿名類別，**條款 18**）中，後者的惟一目標就是終結其外圍實體（enclosing instance）。這種 anonymous class 的實體我們稱爲一個 **finalizer guardian**（終結式守護者），將針對外圍（enclosing）class 的每一個實體被創建出來。外圍實體在一個 private *instance* 欄位中儲存著一個 reference，指向 finalizer guardian，於是 finalizer guardian 在外圍實體存在的同時也合格存在。當 guardian（守護者）被終結，它（應該）執行外圍實體所盼望的終結動作，就好像其 finalizer 是一個位於 enclosing class（外圍類別）內的函式似的：

```
// Finalizer Guardian idiom
public class Foo {
  // Sole purpose of this object is to finalize outer Foo object
  private final Object finalizerGuardian = new Object() {
    protected void finalize() throws Throwable {
      // Finalize outer Foo object
      ...
    }
  };
  ... // Remainder omitted
}
```

注意，上面的 public class `Foo` 並沒有 finalizer（如果不把繼承自 `Object` 的那個無關痛癢的 finalizer 計算在內的話），所以它不在乎其 subclass finalizer 有無呼叫 `super.finalize()`。每一個擁有 finalizer 的 nonfinal public class 都應該認真考慮這項技術。

總之，儘量不要使用 finalizers，除非以它做爲安全網，或是爲了終結非關鍵的原生資源（noncritical native resources）。在需要使用 finalizer 的一些罕見例子中，請記得呼叫 `super.finalize()`。最後一點，如果你需要爲某個 public nonfinal class 撰寫一個 finalizer，試著考慮使用 finalizer guardian 技術，確保 finalizer 被執行，那麼即使 subclass finalizer 不呼叫 `super.finalize()`也沒有關係。

3

通用於所有物件的函式

Methods Common to All Objects

雖然 `Object` 是個 concrete class，其主要設計目的卻是用於擴展（譯注：通常 abstract class 的目的才是用於擴展），因爲它的所有 nonfinal 函式（`equals`, `hashCode`, `toString`, `clone` 和 `finalize`）的設計初衷就是爲了將來被覆寫（overridden），所以它們都有明確的通用契約（general contracts）。任何 class 如果覆寫了這些函式，便有義務遵守這些契約，否則將妨礙與其他「遵守契約的 classes」進行正常合作。

本章告訴你何時以及如何覆寫 `Object` 的 nonfinal 函式。本章不再討論 `finalize` 函式，因爲**條款 6** 已經討論過它。`comparable.compareTo()`不是 `Object` 的函式，但它具有類似性質，因此也在本章討論之列。

條款 7：覆寫 `equals()`時請遵守通用契約（general contract）

覆寫 `equals()`似乎很簡單，但很多方式都是錯誤的，經常導致災難性的後果。避免出錯的最簡單作法是根本不覆寫 `equals()`，這種情形下每個實體只與自身相等。只要下列條件之一得到滿足，那麼做就是合理的：

- class 的每個實體具有本質惟一性（inherently unique）。代表「行爲實物」（active entities）的 classes 就是如此，例如 `Thread`。對這些 classes 而言，`Object` 提供的 `equals()`函式行爲完全正確。
- 你不在意 class 是否提供邏輯相等性（logical equality）。例如 `java.util.Random` 可以覆寫 `equals()`以檢驗兩個 `Random` 實體是否產生同樣的隨機序列，但設計者不認爲用戶需要或期待這個功能，這種情況下繼承 `Object` 的 `equals()`便已足夠。

- 某個 superclass 已覆寫 `equals()`，而從該 superclass 繼承而來的行爲又已適任。例如大多數 `Set` 實作品的 `equals()`繼承自 `AbstractSet`，`List` 的 `equals()`繼承自 `AbstractList`，`Map` 的 `equals()`繼承自 `AbstractMap`。
- 你的 class 是 private 或 package-private，而你確定其 `equals()`永遠不會被呼叫。當然啦我們也可以認爲在這種場合下 `equals()`應當被覆寫如下，以防日後被意外呼叫：

```
public boolean equals(Object o) {
    throw new UnsupportedOperationException();
}
```

那麼，什麼時候應當覆寫 `Object.euqals()`呢？如果 class 具有邏輯相等性（logical equality），而不僅僅只看物件是否完全一致（object identity），而且其 superclass 尚未覆寫 `equals()`實現出它所期望的行爲，這樣的 classes 通常稱爲 value classes，例如 `Integer` 和 `Date`。程式員藉由 `equals()`比較 *reference* to value object 時，希望知道的是它們是否邏輯相等，並不計較它們是否引用（指向）同一個物件。覆寫 `equals()`不僅可以迎合程式員這樣的期望，也使 class 實體得以被用來做爲 map 鍵值或 set 元素並使 map 或 set 具有可預測的、令人滿意的行爲。

有一種不需覆寫 `equals()`的 values class 是 ***typesafe enum***（**條款 21**）。由於這種列舉型別保證每個值最多只有一個物件與之對應，所以 `Object equals()`就足以用來處理這些 classes 的邏輯相等性。

覆寫 `equals()`時，務請遵守其通用契約。以下契約引自 `java.lang.Object` 規格：

`equals()`用以實現等價關係（equivalence relation），並具有以下性質：

- 反身性（reflexive）：對於任意 reference `x`，`x.equals(x)`必定傳回 `true`。
- 對稱性（symmetric）：對於任意 reference `x,y`，若且唯若（*if and only if*）`y.equals(x)`傳回 `true`，則 `x.equals(y)`傳回 `true`。
- 遞移性（transitive）：對於任意 reference `x,y,z`，如果 `x.equals(y)`傳回 `true` 且 `y.equals(z)`傳回 `true`，則 `x.equals(z)`必定傳回 `true`。
- 一致性（consistent）：對於任意 reference `x,y`，多次呼叫 `x.equals(y)`將始終如一地傳回 `true` 或 `false` — 前提是用於 `equals()`比較動作之物件資訊不曾被改動。
- 對於任意 non-null reference `x`，`x.equals(null)`必定傳回 `false`。

除非你是數學偏好者，否則上述文字看起來有點可怕，但你不可忽視它！如果違反了它，你的程式行爲很可能不正常甚至崩潰，而且很難斷定問題的源頭。正如 John Donne 所意喻的那樣，沒有一個 class 是孤島（*no class is an island*；譯注：改編自英國詩人 John Donne 的名句 *no man is an island*），class 實體時常被傳遞給其他 class 實體。因此，包括所有 collections（群集）在內的許多 classes，都指望它們接收的物件確實遵守上述的 `equals()`契約。

既然你知道違反 `equals()`契約的後果，讓我們仔細審視一遍這份契約。好消息是它表面上看起來不太複雜。一旦領會了它，遵守它也就不是難事。讓我們依次考察上述五項條件。

反身性（**Reflexivity**）— 這款條件表示「物件必須與自身相等」。難以想像什麼情況下會無心違反這項條件。如果你違反了它，而且你的 class 實體被加入一個 collection 內，這個 collection 的 `contains()`幾乎肯定會宣稱它並未包含你所加入的那個實體。

對稱性（**Symmetry**）— 這款條件表示「兩物件必須對於彼此是否相等取得一致」。和第一款條件不同的是，無意間違反這一款的情形並不難想像。考慮以下 class：

```
/**
 * Case-insensitive string. Case of the original string is
 * preserved by toString, but ignored in comparisons.
 */
public final class CaseInsensitiveString {
   private String s;

   public CaseInsensitiveString(String s) {
      if (s == null)
         throw new NullPointerException();
      this.s = s;
   }

   // Broken - violates symmetry!
   public boolean equals(Object o) {
      if (o instanceof CaseInsensitiveString)
         return s.equalsIgnoreCase(
            ((CaseInsensitiveString)o).s);
      if (o instanceof String)  // One-way interoperability!
         return s.equalsIgnoreCase((String)o);
      return false;
   }
   ...  // Remainder omitted
}
```

在這個 class 中，立意良好的 `equals()`天真地試圖與一般字串交互操作。假設我們有一個不分大小寫的字串和一個一般字串：

```
CaseInsensitiveString cis = new CaseInsensitiveString("Polish");
String s = "polish";
```

一如預期，`cis.equals(s)`傳回 `true`。問題是雖然 `CaseInsensitiveString` 中的 `equals()`認得一般字串，但 `String` 的 `equals()`卻對區分大小寫的字串茫然無知。於是造成 `s.equals(cis)`傳回 `false`，違反了對稱性。如果你將一個區分大小寫的字串放入一個 collection 內：

```
List list = new ArrayList();
list.add(cis);
```

這時 `list.contains(s)`會傳回什麼？天知道！Sun 目前的實作品傳回 `false`，但這只不過是眾多實作品之一而已。其他實作品或許傳回 true，或者拋出執行期異常。一旦你違反 `equals()`契約，你無法預知其他物件面對你的物件時有什麼表現。

要解決這個問題，只需打消「藉由 `equals()`與 `String` 進行交互操作」這一不切實際的想法。你可以重新構築這個函式，只有單一（譯註：而非如上頁所示的兩個）`return` 動作：

```
public boolean equals(Object o) {
   return o instanceof CaseInsensitiveString &&
      ((CaseInsensitiveString)o).s.equalsIgnoreCase(s);
}
```

遞移性（**Transitivity**）— `equals()`契約中的第三款條件表示，如果物件 A 等於物件 B，而物件 B 又等於物件 C，那麼物件 A 必定等於物件 C。無意間違反這個條件的情形也並不難想像。考慮這種情況：程式員建立了一個 subclass，並在其中增加一個新東西。從另一個角度說，subclass 增加了一些資訊，從而影響了 `equals()`的行爲。讓我們從簡單的不可變（*immutable*）二維 point class 說起：

```
public class Point {
   private final int x;
   private final int y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }

   public boolean equals(Object o) {
      if (!(o instanceof Point))
         return false;
      Point p = (Point)o;
      return p.x == x && p.y == y;
   }

   ... // Remainder omitted
}
```

假設你打算擴展這個 class，爲 `Point` 添加顏色：

```
public class ColorPoint extends Point {
   private Color color;

   public ColorPoint(int x, int y, Color color) {
      super(x, y);
      this.color = color;
   }

   ... // Remainder omitted
}
```

現在，`equals()`應該怎麼寫呢？如果你完全省略 `equals()`，你將會從 `Point` 繼承其函式實作碼，那就未能比較顏色。雖然這並沒有違反 `equals()`契約，但用戶顯然不能接受這種安排。如果你撰寫了一個 `equals()`，令它只在接受「具有相同位置和顏色」的另一個 `ColorPoint` 時才返回 `true`：

```
// Broken - violates symmetry!
public boolean equals(Object o) {
   if (!(o instanceof ColorPoint))
      return false;
   ColorPoint cp = (ColorPoint)o;
   return super.equals(o) && cp.color == color;
}
```

這個函式的問題在於，將一個 `Point` 和一個 `ColorPoint` 拿來比較的所得結果，和反向比較所得結果並不一致。前者忽略了顏色，後者則因引數型別不符而傳回 `false`。爲了更具體說明，讓我們建立一個 `Point` 物件和一個 `ColorPoint` 物件：

```
Point p = new Point(1, 2);
ColorPoint cp = new ColorPoint(1, 2, Color.RED);
```

此時 `p.equals(cp)`傳回 `true`，`cp.equals(p)`傳回 `false`。也許你會試圖透過混合比較（mixed comparisons）— 讓 `ColorPoint.equals()`忽略顏色 — 來糾正這個錯誤：

```
// Broken - violates transitivity.
public boolean equals(Object o) {
    if (!(o instanceof Point))
        return false;

    // If o is a normal Point, do a color-blind comparison
    if (!(o instanceof ColorPoint))
        return o.equals(this);

    // o is a ColorPoint; do a full comparison
    ColorPoint cp = (ColorPoint)o;
    return super.equals(o) && cp.color == color;
}
```

此法提供了對稱性（symmetry），卻喪失了遞移性（transitivity）：

```
ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1, 2);
ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);
```

這時候 `p1.equals(p2)`和 `p2.equals(p3)`都傳回 `true`，`p1.equals(p3)`則傳回 `false`，明顯違反遞移性。前兩次比較忽略了顏色，第三次比較則將顏色納入考慮。

什麼才是解答？這個問題被認爲是物件導向語言中的「等價關係」（equivalence relations）的基本問題。對於一個可實體化的（instantiable）class，並不存在一個「既可增加新外觀（aspect）以擴展 class，又能維持 `equals()`契約」的辦法（譯註：這裡所謂外觀, aspect，指的是欄位）。然而山不轉路轉，我們可以繞道而行。依照**條款 14** 的建議，讓我們優先考慮複合（composition），然後才考慮繼承（inheritance）。我們不再令 `ColorPoint` 擴展 `Point`，而是爲它增加一個 private `Point` 成員，並提供一個 public `view()`（**條款 4**），負責傳回一個 `Point`，代表 `ColorPoint` 的位置。

```
// Adds an aspect without violating the equals contract
public class ColorPoint {
   private Point point;
   private Color color;

   public ColorPoint(int x, int y, Color color) {
      point = new Point(x, y);
      this.color = color;
   }

   /**
    * Returns the point-view of this color point.
    */
   public Point asPoint() {
      return point;
   }

   public boolean equals(Object o) {
      if (!(o instanceof ColorPoint))
         return false;
      ColorPoint cp = (ColorPoint)o;
      return cp.point.equals(point) && cp.color.equals(color);
   }

   ...  // Remainder omitted
}
```

Java 程式庫中，有些 classes 藉由「增加外觀（aspect）」而成爲某個「可實體化 class」的 subclass。例如 `java.sql.Timestamp` 是 `java.util.Date` 的 subclass，它增加了一個 `nanoseconds` 成員。`Timestamp` 的 `equals()`函式違反了對稱性：如果 `Timestamp` 和 `Date` 物件被置於同一個 collection 中，或以其他方式被混用，會導致無法預測的奇怪行爲。`Timestamp` class 有個聲明，警告程式員切勿混用 `Dates` 和 `Timestamps`。雖然不混用它們就不會遇到麻煩，但卻沒有任何方法可以禁止你混用它們，隨之而來的後果就是艱難地除錯。是的，`Timestamp` class 是個特例，你不應效仿它。

注意，你可以爲 abstract class 的 subclass 增加外觀（aspect，意指欄位）而不違反 `equals()`契約。這一點對於根據**條款 20**「以 classes 繼承體系取代 unions」而構造出來的 classes 繼承體系很重要。例如，假設你有一個無任何外觀（aspect）的 abstract class `Shape`，並且一個具有 `radius` 成員的 `Circle` subclass，和一個具有 `Length` 及 `Width` 成員的 `Rectangle` subclass。由於你絕不可能爲此 superclass 生成一個實體，所以剛才提到的問題不會發生。

一致性（**Consistency**）— `equals()`契約的第 4 款條件表示，如果兩個物件相等，除非其中一個（或兩個）被改動，否則它們應該始終保持相等。這一條款算不上是要求，只能說是個提示，因爲 *mutable*（可變）物件可在不同時刻相等於不同的物件，而 *immutable*（不可變）物件不可以。當你撰寫 class 時，請仔細斟酌它是否爲 *immutable*（**條款 13**）。如果你認爲它是，就得確保你的 `equals()`遵守這個契約：相等的物件總是相等，不等的物件總是不等。

有效性（**non-nullity**）— 最後一款條件沒有堂皇的名稱，姑且稱爲「有效性」，表示所有物件不能與 `null` 相等。雖然難以想像當我們呼叫 `o.equals(null)`時意外傳回 `true`，卻不難想像它會意外拋出 `NullPointerException`。`equals()`契約不允許此事發生。許多 classes 都在其 `equals()`函式內明確測試 `null`：

```
public boolean equals(Object o) {
    if (o == null)
        return false;
    ...
}
```

這個測試並非絕對必要。爲檢驗傳入的引數，`equals()`必須先將引數轉爲一個適當型別，這麼一來其存取式（accessors）才可以被喚起，其成員才可以被存取。轉換型別之前，`equals()`必須運用 `instanceof` 運算子檢驗其引數是否爲正確型別：

```
public boolean equals(Object o) {
    if (!(o instanceof MyType))
        return false;
    ...
}
```

如果沒有上述的型別檢查，而傳給`equals()`的引數又屬於錯誤型別，那麼`equals()`會拋出 `ClassCastException`，這就違反了 `equals()`契約。如果 `instanceof` 運算子的第一運算元是 `null`，不論第二運算元是什麼型別，按規矩 `instanceof` 都將傳回 `false` [JLS, 15.19.2]。因此如果 `null` 被傳進來，型別檢驗會傳回 `false`，你也因此不需要獨立一行 `null` 檢驗動作。總結前面的心得，以下是撰寫高品質`equals()`的要點：

1. 以 == 運算子檢查「引數是否爲物件自身的 reference」。如果是，傳回 true。雖然這僅是效率上的一種最佳化手法，但對於可能極爲費時的比較動作而言，還是值得的。
2. 以 instanceof 運算子檢查引數是否爲正確型別。如果不是，就傳回 false。通常「正確型別」就是「equals()函式所屬型別」。偶爾情況下，正確型別會是「這個 class 所實現的某個 interface」。如果有個 interface 強化了 equals() 契約，使我們得以比較「實現這個 interface」的任何 classes，那麼正確型別就應該是該 interface。collection interfaces Set、List、Map 和 Map.Entry 都具有這個特性。
3. 將引數轉換爲正確型別。由於這個轉換在 instanceof 測試之後發生，所以肯定是成功的。
4. 對於 class 內每一個有意義的欄位（fields），檢驗引數中的該欄位是否與物件內的對應欄位吻合。如果所有測試都成功，就傳回 true；否則傳回 false。如果步驟 2 所用型別是個 interface，你必須透過 interface 函式存取引數的有意義欄位；如果步驟 2 所用型別是個 class，你或許可以直接存取那些欄位 —實際情況取決於它們的可存取性（accessibility）。對於非 float 亦非 double 的基本型欄位，應以 == 運算子進行比較；對於 object reference 欄位，應遞迴呼叫 equals()；對於 float 欄位，應先運用 Float.floatToIntBits() 轉換爲 int 值，再以 == 運算子比較兩個 int 值；對於 double 欄位，應先運用 Double.doubleToLongBits 轉換爲 long 值，再以 == 運算子比較兩個 long 值（特殊對待 float 和 double 欄位是有必要的，因爲存在著 Float.NaN、-0.0f 及類似的 double 常量；詳情請閱 Float.equals 說明文件）。對於 array 欄位，應將上述原則施行於每一個元素。某些 object reference 欄位或許可以合法擁有 null 值，爲避免莽撞拋出 NullPointerException，請採用以下的慣用手法加以比較：

   ```
   (field == null ? o.field == null : field.equals(o.field))
   ```

   下面是另一種替代手法，如果 field 和 o.field 經常是同一個 object reference，這種手法的比較速度會更快一些：

   ```
   (field == o.field || (field != null && field.equals(o.field)))
   ```

   對某些 classes（例如先前提到的 CaseInsensitiveString）而言，相對於簡單的「相等性測試」，其欄位比較更複雜些。果真如此，應該可以從 class 的規格書中一目了然。這時候你也許希望在每個物件中保存一個標準型式

（canonical form）；譯注：意指去除贅餘資訊後的精簡算式），如此一來便可對這些標準型式進行省時的精確比較，而非耗時的非精確比較。由於標準型式必須緊緊跟隨物件變化，這項技術最適用於 *immutable* classes（**條款 13**）。

`equals()`的效率有可能受到欄位比較次序的影響。爲了獲得最佳效率，應當首先比較那些「最可能不一致的」或「開銷最小的」欄位，兩者兼具更好。千萬不要將不屬於物件邏輯狀態的欄位（例如 `Object` 中用於同步操作的欄位）也納入比較。贅餘欄位（亦即「可從其他有意義的欄位計算而得者」）不一定要比較，但如果你比較它們，或許會提高 `equals()`的效率。如果某個贅餘欄位相當於整個物件的概括描繪，那麼，比較這個欄位，可以節省你對實際資料所進行的昂貴比較。

5. 當你完成 `equals()`的撰寫，反問自己三個問題：它有對稱性嗎？它有遞移性嗎？它有一致性嗎？（其他兩個特性通常可以自我滿足）。如果答案是"no"，請找出無法保持那些特性的原因，據以修改之。

條款 8 的 `PhoneNumber.equals()`便是依照以上要點構築出來的具體實例。以下是最後數點注意事項：

- `hashCode()`總是應該和 `equals()`一同被覆寫（**條款 8**）。
- 不要賣弄小聰明。如果只是簡單地測試欄位相等性，堅守 `equals()`契約並不困難。如果你過分追求所謂的等價（equivalence），則容易陷入困境。通常，將各式各樣的別名形式（form of aliasing）都予以考慮就不是個好主意。例如 `File` class 不應將「指涉（referring）同一個檔案」的多個符號鏈（symbolics links）視作相等。謝天謝地，它沒有那麼做。
- 不要撰寫出倚賴「不可倚賴資源（unreliable resource）」的 `equals()`。一旦如此，就很難滿足一致性要求。例如 `java.net.URL` 的 `equals()`倚賴 URL 中的主機 IP 位址進行比較。將一個主機名稱轉譯爲 IP 位址，可能需要進行網絡存取，而我們無法保證在不同的時間點獲得相同的結果。這會導致 URL 的 `equals()`違反通用契約，事實上它的確已經在實際運用中引發了一些問題（不幸的是這種行爲由於相容性的需求而無法獲得改善）。除了極少數例外，`equals()`應該對駐留記憶體內的物件執行確定性計算（deterministic computations）。

- 不要在 `equals()`宣告式中以其他型別代替 `Object`。常見的情況是，某位程式員撰寫了一個類似下面的 `equals()`，然後對它無法正常運作百思不解，一籌莫展：

  ```
  public boolean equals(MyClass o) {
  ...
  }
  ```

  問題在於，這個函式並未覆寫（*override*）引數型別爲 `Object` 的 `Object.equals()`，而是重載（*overload*）了它（**條款 26**）。在平常的 `equals()`之外多提供一個這樣的強型別（strongly typed）`equals()`是可以的，只要兩個函式傳回一致結果。但是沒有令人信服的理由讓我們這麼做。某些場合下這麼做也許會提高少許效率，但爲此增加複雜度其實是不值得的（**條款 37**）。

譯註：以下補充 `java.lang.Object` 的部分源碼，展示本章所討論的 `Object` 函式，並補充數個有趣的 interface 源碼：

```
public class Object {
    ...
    public final native Class getClass();
    public native int hashCode();
    public boolean equals(Object obj) {  return (this == obj); }
    protected native Object clone()
       throws CloneNotSupportedException;
    public String toString() {
         return getClass().getName() + "@" +
          Integer.toHexString(hashCode());
    }
    protected void finalize() throws Throwable { }
}
public interface Cloneable {          // 其內空無一物
}
public interface Serializable {       // 其內空無一物
}
public interface Comparable {
   public int compareTo(Object o);
}
```

條款 8:覆寫 equals()時請總是一併覆寫 hashCode()

有些常見臭蟲起源於你未曾在程式中覆寫 hashCode()。是的，任何 classes 如果覆寫了 equals()，便應該一併時覆寫 hashCode()。如果你忘記這麼做，會違反 Object.hashCode()的通用契約並嚐到苦果，從而妨礙你的 class 和所有 "hash-based collections"（包括 HashMap, HashSet, Hashtable）協作時有正確的表現。

以下契約（contract）引自 java.lang.Object 規格書：

- 在同一個應用程式執行期間，對同一個物件呼叫 hashCode()，必須傳回相同的整數結果 — 前提是 equals()所比較的資訊都不曾被改動過。至於同一個應用程式在不同執行期所得的呼叫結果，無需一致。
- 如果兩個物件被 equals(Object)函式視爲相等，那麼對這兩個物件呼叫 hashCode()必須獲得相同的整數結果。
- 如果兩個物件被 equals(Object)函式視爲不相等，那麼對這兩個物件呼叫 hashCode()不必產生不同的整數結果。然而程式員應該意識到，對不同物件產生不同的整數結果，有可能提升 hash table 的效率。

一旦你忘了改寫 hashCode()，你違反的是上述第二款：相等的物件必須具有相等的 hash 碼。對 class 的 equals()而言，兩個獨立實體有可能邏輯上相等，但對 Object hashCode()而言，兩個獨立實體就是兩個毫無共通性的物件，所以 Object 的 hashCode()對兩個獨立實體傳回看似隨機的數值，而非上述契約所要求的兩個相等值。

舉個例子，考慮以下簡化後的 PhoneNumber class，其 equals()乃根據**條款 7** 的要點建構而成：

```
public final class PhoneNumber {
   private final short areaCode;
   private final short exchange;
   private final short extension;

   public PhoneNumber(int areaCode, int exchange,
                      int extension) {
      rangeCheck(areaCode,   999, "area code");
      rangeCheck(exchange,   999, "exchange");
      rangeCheck(extension, 9999, "extension");
```

```
        this.areaCode  = (short) areaCode;
        this.exchange  = (short) exchange;
        this.extension = (short) extension;
    }

    private static void rangeCheck(int arg, int max,
                                   String name) {
        if (arg < 0 || arg > max)
          throw new IllegalArgumentException(name +": " + arg);
    }

    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;
        return pn.extension == extension &&
               pn.exchange  == exchange  &&
               pn.areaCode  == areaCode;
    }

    // No hashCode method!

    ... // Remainder omitted
}
```

假設你企圖使用這個 class 並搭配一個 HashMap：

```
Map m = new HashMap();
m.put(new PhoneNumber(408,867,5309), "Jenny");
```

之後，你或許希望 m.get(new PhoneNumber(408,867,5309))傳回 "Jenny"，但它傳回的卻是 null。請注意這裡涉及兩個 PhoneNumber 實體：一個用來安置於 HasMap 內，與之相等的第二個實體則（試圖）用於檢索。由於 PhoneNumber class 並未覆寫 hashCode()，導致兩個相等實體擁有不同的 hash 碼，違反了 hashCode()契約，導致 get()在某個「與 put()操作對象（某個 hash bucket）並不相同」的物件中搜尋電話號碼。這個問題很容易修正，只要爲 PhoneNumber class 提供一個適當的 hashCode()就行了。

那麼 `hashCode()`看起來應該怎樣呢？撰寫合法但拙劣的函式，是程式員之間司空見慣的事。下面的例子總是合法，卻一點也不高明，你不應該這麼寫：

```
// The worst possible legal hash function - never use!
public int hashCode() { return 42; }
```

它是合法的，因爲它保證相等的物件具有相同的 hash 碼。但它也是差勁的，因爲它使每個物件都具有相同的 hash 碼，這會造成每一個物件都被 *hashing* 置入同一個 hash bucket，致使 hash tables 蛻變爲 linked lists（譯註：如果考慮 *rehashing* 機制，還是能夠打散開來）。原本只需線性執行時間，如今卻需要二次方執行時間，這對大型 hash table 而言會影響程式的實用性。

優秀的 hash 函式往往面對不等物件產生不同的 hash 碼。這正是 `hashCode()`契約中第三款條文的含意。理想的 hash 函式應當爲「所有不等物件所組成的合理群集（reasonable collection）」均勻分配所有可能的 hash 值（碼）。要獲得這個理想目標非常不容易。幸運的是獲得一個近似目標並不太困難。下面是一些要點：

1. 將一個非 0 常數，例如 17，儲存於 `int result` 變數中。
2. 對物件中的每一個有意義的欄位 `f`（更確切地說是被 `equals()`所考慮的每一個欄位）進行如下處理：
   A. 對這個欄位計算出型別爲 `int` 的 hash 碼 `c`：
      i. 如果欄位是個 `boolean`，計算`(f ? 0 : 1)`。
      ii. 如果欄位是個 `byte,char,short` 或 `int`，計算`(int)f`。
      iii. 如果欄位是個 `long`，計算`(int)(f^(f >>> 32))`。
      iv. 如果欄位是個 `float`，計算 `Float.floatToIntBits(f)`。
      v. 如果欄位是個 `double`，計算 `Double.doubleToLongBits(f)`，然後將計算結果按步驟 2.A.iii 處理。
      vi. 如果欄位是個 object reference，而且 class 的 `equals()`透過「遞迴呼叫 `equals()`」的方式來比較這一欄位，那麼就同樣也對該欄位遞迴呼叫 `hashCode()`。如果需要更複雜的比較，請對該欄位運算一個標準表述式（canonical representation），並對該標準表述式呼叫 `hashCode()`。如果欄位值是 `null`，就傳回 0（或其他常數；傳回 0 是傳統做法）。

vii. 如果欄位是個 array，請將每個元素視爲獨立欄位。也就是說對每一個有意義的元素施行上述規則，用以計算出 hash 碼，然後再依步驟 2.B 將這些數值組合起來。

B. 將步驟 A 計算出來的 hash 碼 `c` 按下列公式組合到變數 `result` 中：

```
result = 37*result + c;
```

3. 傳回 `result`。

4. 完成 `hashCode()`之後，反躬自省一下：是否相等的實體具有相等的 hash 碼？如果不是，找出原因並修正問題。

這份契約允許你將贅餘欄位（redundant fields。譯註：可由其他欄位計算而得的欄位）排除在 hash 碼計算之外。換言之它允許你排除任何可經由「已被含入整個計算之中」的欄位(s)所導出的欄位。至於「`equals()`函式未用到的欄位」一定得排除。如果你沒有排除那些欄位，可能導致違反 `hashCode()`契約的第二款條文。

步驟 1 用到了一個非 0 初值，因此最終的 hash 值會受到「hash 值（步驟 2.A 的計算結果）爲 0」的欄位的影響（譯注：這樣是好的）。如果步驟 1 以 0 爲初值，最終的 hash 值便不會受到這種欄位的任何影響（譯注：因爲 0 乘以任何數值的結果都是 0），從而增加碰撞機率（譯注：這是不好的）。數值 17 乃隨意指定。

步驟 2.B 中的乘法使得 hash 值與欄位順序有關。因此如果 class 包含多個相似欄位，這樣的安排會導出更好的 hash 函數。例如，如果對 `String` 的 hash 函數省略這個乘法，那麼擁有相同字元但字元順序不同的所有字串將擁有完全相同的 hash 碼。以 37 爲乘數乃因爲它是個奇質數。如果採用偶數而且乘法滿溢（overflowed），資訊將會佚失，因爲乘以 2 相當於移位（shifting）操作。採用質數的好處在這裡並不是那麼一目了然，但傳統以來都在此處使用質數。

讓我們將這個辦法應用於 `PhoneNumber` class，它有三個有意義的欄位，都是 `short` 型別。直接套用上述要點，產生下面的 hash 函式：

```
public int hashCode() {
    int result = 17;
    result = 37*result + areaCode;
    result = 37*result + exchange;
    result = 37*result + extension;
    return result;
}
```

這個函式傳回一個簡單明確的結果，其輸入參數就是 PhoneNumber 實體中的三個有意義的欄位。顯而易見，相等的 PhoneNumber 實體一定有著相等的 hash 碼。這個函式實際上是 PhoneNumber class 的一個相當出色的 hashCode()函式實作碼，可以和 Java 程式庫 1.4 版中的實作品相媲美。它不僅簡潔、速度快，而且有能力把不同的電話號碼分配給不同的 hash bucket。

如果你的 class 不可變（*immutable*），而且計算 hash 碼的代價甚爲可觀，你或許會考慮將 hash 碼儲存（caching）於物件之中待用，而不是每次被索求後重新計算。如果你認爲某型別的大多數物件將被作爲 hash 鍵，那麼你應當在實體創建時就計算 hash 碼，否則便應該在 hashCode()第一次被呼叫時進行惰式初始化（lazily initialize，**條款 48**）。我們的 PhoneNumber class 不見得受益於這個方案，這裡只是告訴你如何著手：

```
// Lazily initialized, cached hashCode
private volatile int hashCode = 0;  // (See Item 48)

public int hashCode() {
    if (hashCode == 0) {
        int result = 17;
        result = 37*result + areaCode;
        result = 37*result + exchange;
        result = 37*result + extension;
        hashCode = result;
    }
    return hashCode;
}
```

雖然，遵照本條款所列要點便可以產生相當不錯的 hash 函式，但無法因此產出最高水平的hash函式 — 事實上Java程式庫1.4版也沒有提供這樣水平的hash函式。高水平 hash 函式是一個十分活躍的研究領域，應該由數學家和計算機理論學家去研究。或許 Java 平台的後繼版本會爲其 classes 提供更高水平的 hash 函式，並提供工具函式（utility methods）讓一般程式員也能夠建構這樣的 hash 函式。本條款所描述的技術對大部分應用而言應該已經足夠。

在 hash 碼的計算過程中，不要爲了提升效率而冒險。我的意思是，不要排除物件中有意義的成分。雖然這麼做所產生的 hash 函式或許能夠執行更快速一些，但其品質或許會使 hash table 退化至難以接受的地步（譯註：如果碰擊機率大增的話）。更具體地說，實際應用時 hash 函式可能會面臨大量「在你刻意忽略的領域中仍有很大差異」的物件實體。如果這種情況發生，上述所說的那種 hash 函式會把所有

實體映射（*map*）到極少量的數個 hash 碼上，那麼 hash-based collections 將表現出二次方時間效率（quadratic performance）。這並不是杞人憂天；在 1.2 版之前的所有 Java 平台上，`String` 的 hash 函式最多只審查 16 個字元（這 16 個字元是從第一個字元起在整個字串上均勻選取），面對大量諸如 URLs 之類的層狀名稱（hierarchical names），這樣的 hash 函式會表現出類似上述所提的病態行爲。

Java 程式庫中的許多 classes 如 `String`、`Integer` 和 `Date`，都將它們的 `hashCode()` 所傳回的確切值（exact value）指定爲實體值（instance value）的某個函數。這通常不是個好主意，因爲這會嚴重限制 hash 函式在未來版本中的改善空間。如果你不指定 hash 函式的實作細節，那麼一旦發現 hash 函式有缺陷，就可以在下一版本中修正，不必擔心破壞程式庫與「密切倚賴 hash 函式所傳回之確切值」的客戶之間的相容性。

條款 9：總是覆寫 toString()

雖然 java.lang.Object 提供了一份 toString()實作品，但它傳回的 string（字串）往往不是你的 class 用戶者所期望的樣子。它包含 class 名稱，後接一個 '@' 符號和一個不帶正負號的十六進制 hash 碼，例如"PhoneNumber@163b91"。根據 toString()通用契約的說法，傳回的 string 應當是一段「簡明扼要、資訊豐富、易被人類閱讀」的表達文字。就算我們姑且認爲 "PhoneNumber@163b91" 簡明易讀吧，但是和 "(408)867-5309" 相比卻未必具備豐富的提示資訊。toString()通用契約建議所有 subclasses 都覆寫該函式。這實在是個饒富價值的忠告。

儘管不像遵守 equals()契約和 hashCode()契約那樣重要（**條款 7** 和**條款 8**），提供一個出色的 toString()函式還是會吸引別人更樂於使用你的 class。當你的物件傳遞給 println()、字串接合運算子（+）或 1.4 版的 assert()時，toString()會自動被喚起。如果你提供一個出色的 toString()，很容易（像下面這樣）生成一段有益的診斷訊息：

```
System.out.println("Failed to connect: " + phoneNumber);
```

即使你的 class 未曾覆寫 toString()，客戶程式員也可以採用上式生成診斷訊息。但除非你覆寫了 toString()，否則產生出來的訊息難以理解。提供出色的 toString()可使 class 實體乃至於包含實體的 object reference 都受益，尤其是對 collections（群集）而言。當你列印一個 map 時，你願意看到哪一種訊息？是 "{Jenny=PhoneNumber @163b91}" 還是 "{Jenny=(408) 867-5309}"？

在實際應用中，toString()應當傳回物件內令人感興趣的一切資訊，正如上述電話號碼實例所展示的那樣。如果物件太大或包含了不便以 string 表現的狀態，上述建議就難免不切合實際。這種情形下 toString()應該傳回一份概要資訊，例如 "Manhattan white pages (1487536 listings)" 或 "Thread[main,5,main]"。最理想的情況是傳回的 string 帶有自我解釋性（self-explanatory），但 Thread 例子不符合這個要求。

實現 toString()時你必須做一個重要決定：是否在文件中明確說明回返值的格式。我建議你對 value classes（例如電話號碼或矩陣）這麼做。「明確說明格式」的優點在於，它可以被當作一個標準的、明確的、人類可讀的物件表達形式。這

種表示方式可用於輸入、輸出，以及像 XML 文件這種永續性的（persistent）、人類可讀的資料物件中。一旦你指定了格式，再提供一個相應的 `String` 建構式（或一個 static *factory*，見**條款 1**）通常是個好主意，這麼一來程式員可輕易在物件和其「string 表述形式」之間來回轉換。Java 程式庫的許多 value classes，包括 `BigInteger`、`BigDecimal` 和大部分基本型別（primitive types）的外覆類別（wrapper classes），都採用這種作法。

但是「明確說明 `toString()`回返值格式」也不是沒有缺點。它的缺點在於，一旦你這麼做，而且你的 class 被廣泛使用的話，便將永遠被這種格式纏住而無法擺脫。你的 class 用戶有可能撰寫程式分析這個表述格式，生成它，將它嵌入永久資料中。如果你在未來版本中改變了表述格式，你將危及那些用戶的程式和資料，他們會大聲抗議。只要不明確指定格式，你就保留了在後續版本中「增加資訊」或「改善格式」的靈活性。

無論是否「明確指定格式」，你都應該在文件中清楚表明你的意向。如果你指定了格式，更應當那麼做。下面這個 `toString()`用來配合**條款 8** 中的 `PhoneNumber`：

```
/**
 * 本函式傳回電話號碼的「字串表述形式」（string representation）。字串包含
 * 14 個字元，格式爲："(XXX) YYY-ZZZZ"，其中 XXX 表示 area code（區碼），
 * YYY 表示 extension（擴充碼），ZZZZ 表示 exchange（交換碼）。
 * (每個大寫字母代表一個十進制數字。)
 *
 * 如果電話號碼的三個成分中的任何一個無法填滿欄位，
 * 就由前導的 0 填補。假設 extension（擴充碼）的值是 123，那麼
 * 字串表述形式的最後 4 個字母將是 "0123"。
 *
 * 注意，在 area code（區碼）之後的閉合括號和 exchange（交換碼）的第一個
 * 數字之間有個空格，做爲分隔符號。
 */
public String toString() {
   return "(" + toPaddedString(areaCode, 3) + ") " +
          toPaddedString(exchange,  3) + "-" +
          toPaddedString(extension, 4);
}
```

```
/**
 * 將 int 轉換爲指定長度的字串，必要時由前導 0 填補。
 * 假設 i >= 0，1 <= length <= 10，
 * 且 Integer.toString(i).length() <= length。
*/
private static String toPaddedString(int i, int length) {
    String s = Integer.toString(i);
    return ZEROS[length - s.length()] + s;
}

private static String[] ZEROS =
    {"", "0", "00", "000", "0000", "00000",
     "000000", "0000000", "00000000", "000000000"};
```

如果你決定不明確指出格式，文件注釋應當像這樣：

```
/**
 * 傳回值的簡要描述。爲了將來的變化可能，確切之表述形式不做明確指定。
 * 以下可被視爲一種典型形式：
 *
 * "[Potion #9: type=love, smell=turpentine, look=india ink]"
 */
public String toString() { ... }
```

讀了這些注釋之後，那些倚賴「格式細節」來撰寫程式或保存資料的程式員，面對「因格式變化而帶來的影響」，也就只能自扛責任了。

無論是否明確指定格式，爲「含於 `toString()`回傳值中的資訊」提供一種編程途徑，總是個好主意。例如 `PhoneNumber` class 應當包含 area code（區碼）、exchange（交換碼）和 extension（擴充碼）的存取式（accessors）。如果你不這樣做，就是強迫「對這些資訊有所需求」的程式員自行分析 `toString()`傳回的字串。這不但降低效率，逼迫程式員做無謂的工作，而且一旦你改變格式，整個過程也容易出錯並容易導致系統脆弱不堪。即使你已經表明字串格式有可能發生變化，如果不提供存取式，仍然會使它成爲實質上的（*de facto*）API。

條款 10：審慎地覆寫 clone()

譯註：閱讀本條款的同時，建議參考 java.util 的 collection classes 源碼，例如 HashMap 或 ArrayList，觀察它們的 *clone* 作法。

Cloneable interface 旨在做爲物件的 mixin interface（**條款 16**），宣示它們允許 *cloning*（克隆）動作。不幸的是它並沒有實現初衷。主要瑕疵在於它缺乏 clone()函式，而 Object 的 clone()卻又屬於 protected 級別。如果不借助反射機制（reflection，**條款 35**），你無法僅僅因爲一個物件實現了 Cloneable 就對它呼叫 clone()。甚至反射式呼叫（reflective invocation）也可能失敗，因爲無法保證物件一定具有可取用的 clone()。儘管存在這樣那樣的缺點，這個機制還是獲得了十分廣泛的運用，因此值得我們好好理解它。本條款告訴你如何實現一個行爲良好的 clone()，討論什麼時候適合使用它，並簡單探討其替代形式。

既然 Cloneable 不含任何函式，它到底能幹什麼呢？是的，它用來決定 Object 的protected clone()實作碼行爲：如果某個class實現了Cloneable，Object clone()便傳回一個「欄位逐一拷貝」（filed-by-field copy）的副本物件；否則便拋出 CloneNotSupportedException 異常。這是一種極特殊的 interface 用法，你不應仿效。一般說來，「實現某個 interface」表示這個 class 能夠爲其客戶做點什麼，然而實現 Cloneable 卻是改變了 superclass 的 protected 函式行爲（譯註：Serializable 異曲同工，見第 10 章）。

爲了實現 Cloneable interface 並使它對 class 真正產生影響，class 及其所有 superclasses 必須遵守一個相當繁複、非強制性、多數未見諸文件的協定（protocol）。其結果形成了一套超脫語言的機制：在不呼叫建構式的情況下創建一個物件。

clone()通用契約十分薄弱。下面就是其內容，引自 java.lang.Object 規格書：

> 創建並傳回物件的一個複件（copy）。複件的精確意義取決於物件所屬的 class。一般而言，對任意物件 x，以下式子爲 true：
>
> ```
> x.clone() != x
> ```
>
> 而且以下式子爲 true：
>
> ```
> x.clone().getClass() == x.getClass()
> ```
>
> 但這些並非絕對必要。通常情況下，以下式子爲 true：

```
x.clone().equals(x)
```

這也不是絕對必要。拷貝一個物件，典型情況是創建「該物件所屬的 class」的一份新實體，並可能同時要求拷貝內部資料結構。但不得喚起建構式。

這份契約存在一些問題。「不得喚起建構式」這句話太強硬了。一個行爲良好的 `clone()`可以呼叫建構式以創建物件，並於創建後再複製物件內部資料。面對 final class，`clone()`甚至可以傳回由建構式創建的物件。

此外，「`x.clone().getClass()`通常應當與 `x.getClass()`完全相同」這一條文也禁不起推敲。現實世界中，程式員認爲如果他們擴展（繼承）一個 class，並從 subclass 呼叫 `super.clone()`，傳回的物件理應是 subclass 實體（譯註：但其型別得以 superclass 表示）。superclass 惟一得以提供這項功能的辦法就是傳回一個「經由呼叫 `super.clone()`而得」的物件。如果 `clone()`傳回一個「透過建構式創建出來的」物件，其所屬的 class 將是不正確的。所以，如果你覆寫 nonfinal class 的 `clone()`，你應該傳回「經由呼叫 `super.clone()`而獲得」的物件。如果所有 superclasses 都遵守這項規則，層層呼叫 `super.clone()`，最終將喚起 `Object clone()`，從而創建一個型別正確的實體。這個機制有點類似「全自動建構鏈」（automatic constructor chaining），但它不是強制性的。

直至 Java 1.3 版，`Cloneable` interface 都未明確說明其實現者（某個 class）應有的職責。規格書中僅僅描述「如果實現這個 interface，將會以何種方式影響 `Object clone()`的行爲」，其他隻字未提。現實世界中，程式員會設想「`Cloneable` 的實現者應該提供一個適當功能的 public `clone()`」，然而通常並非如此，除非實現者（某個 class）的所有 superclasses 都提供了行爲良好的 `clone()`實作函式；至於這些實作函式是 public 或 protected 倒是無關緊要。

假設你打算爲某個 class 實現 `Cloneable`，而其 superclasses 均有提供行爲良好的 `clone()`。你從 `super.clone()`獲得的物件，或許接近（亦或許不接近）你的最終傳回結果，視 class 的本質（nature）而定。從每一個 superclass 的立場來看，這個物件是原物件的全功能克隆件（full functional clone）。class 所宣告的欄位（如果有的話）的值將與被克隆之正本物件一致。如果每個欄位包含的都是基本型別，或包含一個 *reference* to "immutable object"，傳回的物件可能就是你需要的，無需額外處理。**條款 8** 的 `PhoneNumber` class 就是如此。這種情況下你所要做的僅僅是提供對 `Object` 的 protected `clone()`的 public 訪問途徑：

```
public Object clone() {
    try {
```

```
            return super.clone();
        } catch(CloneNotSupportedException e) {
            throw new Error("Assertion failure");  // Can't happen
        }
    }
```

然而如果你的物件內含 *reference* to "mutable object"，剛才所說的 `clone()`實現方式會帶來災難。考慮**條款 5** 的 `Stack` class：

```
public class Stack {
    private Object[] elements;
    private int size = 0;

    public Stack(int initialCapacity) {
        this.elements = new Object[initialCapacity];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }

    // Ensure space for at least one more element.
    private void ensureCapacity() {
        if (elements.length == size) {
            Object oldElements[] = elements;
            elements = new Object[2 * elements.length + 1];
            System.arraycopy(oldElements, 0, elements, 0, size);
        }
    }
}
```

假設你希望這個 class 是 *cloneable*。如果其 `clone()`僅僅 `return super.clone()`，產生出來的 `Stack` 實體的 `size` 欄位會帶有正確值，但 `elements` 欄位將指向和原（源）`Stack` 實體所指的同一個 array。這種情況下，一旦修改正本物件，便會破壞克隆件的約束條件（invariants；譯注：意指物件欄位之間的邏輯關係，例如 `elements` 的元素個數應該總是等於 `size`），反之亦然。於是你很快便會發現你的程式出現荒謬結果，或拋出一個 `NullPointerException` 異常。

如果在 Stack class 中呼叫建構式，上述情形就絕對不會發生。事實上，clone()就是另一種形式的建構式，你必須確保它無損於源物件，並正確設立克隆件的約束條件（invariants）。爲了讓 Stack clone()能夠有效運作，必須拷貝 stack 的內部構件。達此目的的最簡單辦法就是對著 elements array 遞迴呼叫 clone()：

```
public Object clone() throws CloneNotSupportedException {
    Stack result = (Stack) super.clone();
    result.elements = (Object[]) elements.clone();
    return result;
}
```

注意，如果 elements 欄位是 final 屬性，這個辦法將就不再適用，因爲 clone()無法對這個 final 欄位賦予新值。這是一個基本問題：clone 架構（architecture）與「用以指涉可變物件（*mutable* objects）」之 final 欄位的常規用法不相容，除非此一可變物件可以在物件及其克隆件之間被安全共享。因此，爲了讓 class 成爲 *cloneable*，或許必須去除 class 之中某些欄位的 final 屬性。

別以爲只要遞迴呼叫 clone()就萬事大吉。假設你撰寫某個 hash table 的 clone()，這個 hash table 內部包含一個 buckets array，每個 bucket 指向一個「以 key-value pair 爲元素」的 linked list 的首筆元素（如果 bucket 爲空，就指向 null）。爲了保證效率，class 內部不採用 java.util.LonkedList，而是自我實現了一個輕量級單向 linked list：

```
public class HashTable implements Cloneable {
    private Entry[] buckets = ...;

    private static class Entry {
        Object key;
        Object value;
        Entry  next;

        Entry(Object key, Object value, Entry next) {
            this.key   = key;
            this.value = value;
            this.next  = next;
        }
    }

    ... // Remainder omitted
}
```

假設你只是遞迴克隆（*clone*）bucket array，就像我們在 `Stack` 中的作爲：

```
// Broken - results in shared internal state!
public Object clone() throws CloneNotSupportedException {
    HashTable result = (HashTable) super.clone();
    result.buckets = (Entry[]) buckets.clone();
    return result;
}
```

儘管克隆件本身有其自己的 bucket array，但那個 array 卻指向與源物件相同的 linked list，這很容易引起克隆件和源物件之間不可預期的行爲。爲修正這個問題，你必須逐一拷貝由各個 bucket 組成的各個 linked list。下面是一種常見作法：

```
public class HashTable implements Cloneable {
    private Entry[] buckets = ...;

    private static class Entry {
        Object key;
        Object value;
        Entry  next;

        Entry(Object key, Object value, Entry next) {
            this.key   = key;
            this.value = value;
            this.next  = next;
        }

        // Recursively copy the linked list headed by this Entry
        Entry deepCopy() {
            return new Entry(key, value,
                next == null ? null : next.deepCopy());
        }
    }

    public Object clone() throws CloneNotSupportedException {
        HashTable result = (HashTable) super.clone();
        result.buckets = new Entry[buckets.length];
        for (int i = 0; i < buckets.length; i++)
            if (buckets[i] != null)
                result.buckets[i] =
                    buckets[i].deepCopy();

        return result;
    }
    ... // Remainder omitted
}
```

private class `HashTable.Entry` 擴充支援一個深層拷貝（`deep copy`）函式。`HashTable` 的 `clone()`配置一個新的、具有正確容量的 buckets array，然後迭代處理原本的 buckets array，對每一個不爲空的 bucket 實施深層拷貝。`Entry` 的深層拷貝函式（`deepCopy()`）遞迴呼叫自己，用以拷貝以此 `Entry` 爲首的整個 linked list。雖然這項技術很酷，而且在 buckets 不太長的場合中可以良好運作，但這並非是拷貝 linked list 的優越辦法，因爲它爲 linked list 中的每一個元素耗費了一個 stack frame。一旦 linked list 過長，很容易引發 stack 滿溢（overflow）。爲防止這種事情發生，你可以在 `deepCopy()`中以迭代（iteration）替換遞迴（recursion）：

```
// Iteratively copy the linked list headed by this Entry
Entry deepCopy() {
    Entry result = new Entry(key, value, next);

    for (Entry p = result; p.next != null; p = p.next)
        p.next = new Entry(p.next.key, p.next.value, p.next.next);

    return result;
}
```

克隆（*cloning*）複雜物件的最後一個辦法是呼叫 `super.clone()`，將所生物件的所有欄位設爲原始狀態，然後呼叫較高層（higher-level）函式，重新生成物件狀態（譯注：此法乃是模擬物件當前狀態的完整建立過程，並非僅僅克隆物件的狀態資訊）。對我們的 `HashTable` 例子而言，buckets 欄位應先被初始化爲新的 bucket array，然後對著「被仿造之 hash table 中的每一個 key-value mapping」呼叫 `put(key,value)`（書上未顯示）。這通常可以產生一個簡潔優雅的 `clone()`，但其執行速度與「直接操縱物件及其克隆件的內部構件」相比，稍遜一籌。

和建構式一樣，`clone()`不應該在建構過程中對著克隆件呼叫任何 nonfinal 函式（**條款 15**）。如果 `clone()`呼叫一個被覆寫函式，那麼在「該函式定義所在」之 subclass 有機會修正新物件的狀態之前，該函式就先被執行了。這很可能導致克隆件和原物件之間的不一致。因此上一段所討論的 `put(key,value)`如果不是 final 就應該是 private（如果它是 private，它大概是某個 nonfinal public 函式的輔助函式）。

`Object` 的 `clone()`宣稱「可能拋出 `CloneNotSupportedException` 異常」，但是覆寫後的 `clone()`或許會忽略這個宣示。final classes 的 `clone()`應當忽略這項宣示，因爲人們更願意使用那些不含「可控式異常」（checked exceptions）的函式（**條款 14**；譯注：所謂「可控式異常」是指在函式宣告式中指明的可能被該函

式拋出的異常）。如果一個可擴展的（extendable）class — 尤其是意在繼承的 class（**條款 15**）— 覆寫了 `clone()`，這個 `clone()`應當宣告它自己「可能拋出 `CloneNotSupportedException` 異常」。這麼做便允許 subclass 藉由提供以下 `clone()`函式得體地去除 clone 功能：

```
// Clone method to guarantee that instances cannot be cloned
public final Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
}
```

遵守以上建議並非絕對必要，原因在於「不期望被克隆」的 subclass 即使沒有宣告其所覆寫的 `clone()`可能拋出「可控式異常」`CloneNotSupportedException`，也可以拋出任何「不可控異常」如 `UnsupportedOperationException`。然而通常我們習慣在這種情況下拋出 `CloneNotSupportedException` 異常。

概括而言，所有實現 `Cloneable` 的 classes 都應該以 public 方式覆寫 `clone()`。這個 public 函式應該首先呼叫 `super.clone()`，然後調整需要修改的欄位。通常這意味拷貝任何一個「包含正本物件內部深層結構」之 *mutable*（可變）物件，並將「指向這些物件」的 references 替換爲「指向複本物件」的 references。雖然這些內部構件的複本通常可藉由遞迴呼叫 `clone()`產生，但這並非最佳辦法。如果 class 只含基本型欄位或 references to *immutable* object，多半情況下沒有什麼欄位需要額外修改。但是這項規則存在例外情況，例如一個「表示序號（serial number）或其他惟一識別號」的欄位，或一個「表示物件創建時間」的欄位，即使它們是基本型別或不可變（*immutable*），也需要進一步修改。

真的必須如此複雜嗎？的確如此。一旦你擴展了一個實現 `Cloneable` 的 class，你幾乎就是必須實現一個行爲良好的 `clone()`，否則就必須提供「物件複製」的某種替代方案，或乾脆不提供這項能力。例如令 *immutable* classes 支持「物件複製」就沒有什麼意義，因爲其複本物件實際上無法與正本物件有所區別。

「物件複製」的一個好辦法是提供所謂的 *copy* 建構式。*copy* 建構式是個稍微特別的建構式，接受單一引數，該引數必須隸屬「*copy* 建構式所屬之 class」，例如：

```
public Yum(Yum yum);
```

另一個較不重要的變形是提供一個 static ***factory***（靜態工廠）取代建構式：

```
public static Yum newInstance(Yum yum);
```

copy 建構式和其 static ***factory*** 變形，較 `Cloneable`/`clone()`優勢多多：它們不倚賴一個帶有風險的、超脫語言之外的物件創建機制；它們不要求用戶遵循尚未形成良好文件的規矩；它們不會與 final 欄位的正常運用發生衝突；它們不要求用戶捕捉非必要的「可控式異常」（checked exceptions）；它們爲用戶提供了一個靜態化型別物件（statically typed object）。由於我們不可能將一個 *copy* 建構式或一個 static ***factory***（靜態工廠）放入 interface 之中，所以 `Cloneable` 將因爲缺乏 public `clone()`而無法行使 interface 的職權，因此你不會因爲以 *copy* 建構式代替 `clone()` 而損失了 interface 帶來的機能。

此外，*copy* 建構式（或 static ***factory***）可接受一個引數，其型別爲「class 所實現之適當 interface」。例如所有通用型 collection 實作品，習慣上都提供引數型別爲 `Collection` 或 `Map` 的 *copy* 建構式。這種 interface-based *copy* 建構式允許用戶選擇複製動作的實現方式，不強迫用戶接受正本物件的實現方式。舉個例子，假設你有一個 `LinkedList l`，你打算將它複製爲 `ArrayList`。`clone()`沒有提供這個功能，但是用一個 *copy* 建構式很容易就辦到：`new ArrayList(l)`。

既然前面給出了這麼多 `Cloneable` 相關問題，我們可以有把握地說，任何其他 interface 都不應當擴展（*extend*）`Cloneable`，爲繼承而設計的 class（**條款 15**）也不應當實現（*implement*）它。鑒於它的眾多缺點，有些老練程式員拿定主意絕不覆寫和呼叫 `clone()`，也許只偶爾用它來複製 array。你必須清楚意識到，你至少要爲「爲繼承而設計的 class」提供一個行爲良好的 protected `clone()`，否則 subclass 就不可能實現 `Cloneable`。（譯注：也就是說你必須保證這個 subclass 的所有 superclasses 都具有行爲良好的 `clone()`）

條款 11：考慮實現 Comparable

和本章討論的其他函式不同，compareTo()並非宣告於 Object 之中。它其實是 java.lang.Comparable interface 的惟一函式，性質上和 Object 的 equals()很相近，只不過在單純的「相等比較」之外，它還允許次序比較（order comparisons）。如果某個 class 實現了 Comparable，便是暗示其實體具有內在次序關係（natural ordering）。如果 array 的每一個元素都是「實現了 Comparable」的物件，array 的排序動作非常簡單：

```
Arrays.sort(a);
```

對 *comparable* 物件進行查詢、計算極值、自動維護其「經過排序的群集」（sorted collections）等等，也都一樣容易。下面這段程式利用「String 實現了 Comparable」這一特點，將命令列引數（command-line arguments）按字母排序並去掉重複元素，然後列印出來：

```
public class WordList {
    public static void main(String[] args) {
        Set s = new TreeSet();
        s.addAll(Arrays.asList(args));
        System.out.println(s);
    }
}
```

藉由「實現 Comparable」，你的 class 可與大量「以此 interface 爲基礎」之泛型演算法和 collection 實作品協同工作。極小的努力就可以獲得巨大的能量。事實上 Java 程式庫中的所有 value classes 都實現了 Comparable。如果你正撰寫一個具有明顯內在次序（natural ordering，例如字母順序、數字順序或時間順序）的 value class，強烈建議你實現這個 interface。本條款正是告訴你如何進行這件事。

compareTo()的通用契約性質上和 equals()類似。下面就是其內容，引自 Comparable 規格書：

> 比較某物件與另一指定物件的順序。當這個物件小於、等於、大於指定物件時，分別傳回負整數、0、正整數。如果指定物件的型別不允許與這個物件進行比較，就拋出 ClassCastException 異常。

下面的描述中，符號 `sgn(`*expression*`)`表示數學函數 *signum*，根據 *expression* 爲負數、`0` 或正數，傳回相應的`-1`、`0`、或 `1`。

實作者必須確保，對於所有 `x` 和 `y`，都能夠滿足 `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))`。這意味「`x.compareTo(y)`拋出異常，若且唯若（*if and only if*）`y.compareTo(x)`拋出一個異常」。

- 實作者必須確保數值間的關係具有遞移性，也就是說`(x.compareTo(y))>0 && y.compareTo(z)>0)`意味 `x.compareTo(z)>0`。
- 實作者必須確保運算式 `x.compareTo(y)==0` 意味對任意 `z` 而言，運算式 `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`永遠成立。
- 強烈建議（但不要求）滿足`(x.compareTo(y)==0) == (x.equals(y))`。一般而言，任何實現 `Comparable` interface 卻違反本項條件之某個 class，都應當明確告訴用戶這一事實。以下是被大家推薦的說明方式：「注意：這個 class 的內在次序（natural ordering）與相等關係（equals）並不相容。」

不要被這個契約的數學性質嚇倒。和 `equals()`契約（**條款 7**）一樣，`compareTo()`的契約並不像表面那麼複雜。在 class 內，任何合理的順序關係都自然而然會滿足這個契約。在各 classes 之間，`compareTo()`不再像 `equals()`那樣非得做出某種決斷不可：如果兩個進行比較的 object references 分別指向不同的 class，那就允許拋出 `ClassCastException` 異常。通常這也正是 `compareTo()`在這種場合應有的反應。雖然這份契約不排除「class 與 class 之間」（所謂 interclass）的比較，但直到 1.4 版，Java 程式庫仍然沒有任何 class 支援這種比較。

就像「任何 class 如果違反 `hashCode()`契約，有可能危及其他倚賴 hashing 行爲的 classes」那樣，違反 `compareTo()`契約者，亦有可能危及其他倚賴「比較行爲」的 classes。這樣的 classes 包括 sorted collections（自動排序的群集）、`TreeSet` 和 `TreeMap`、工具類別 `Collections` 和 `Arrays`，它們都內含搜尋和排序演算法。

讓我們審視一下 `compareTo()`契約中的條文。第一條表示，如果你反轉兩個 object references 的比較方向，會發生以下事情：如果第一個物件小於第二個，那麼第二個物件必須大於第一個；如果第一個物件等於第二個，那麼第二個物件必須等於第一個；如果第一個物件大於第二個，那麼第二個物件必須小於第一個。第二條文表示，如果第一個物件大於第二個，而且第二個物件大於第三個，那麼第一個物件必須大於第三個。最後一項條文表示，所有相等物件，與其他物件比較時，必須產生相同的結果。

這三個條文的一個推論結果是，運用 `compareTo()`所做的相等性測試必須遵守 `equals()`所遵守的一切約束：反身性、對稱性、遞移性和有效性。所以 `equals()` 的告誡同樣適用於此：對於一個可實體化（instantiable）class，並不存在一個「既可增加新外觀（aspect）以擴展 class、又能維持 `compareTo()`契約」的辦法（**條款 7**）。`equals()`的迂迴解法在此也同樣適用：如果你打算對一個實現了 `Comparable` 的 class 增加一個有意義的外觀（aspect），請不要擴展（繼承）這個 class，而是另寫一個 class，其中有個欄位包含這個 class（譯注：亦即不使用繼承，改用複合），然後提供一個返回該欄位的 "view"（映像）函式。這種方法使你得以自由實現第二個 class 的 `compareTo()`，同時允許其客戶必要時將第二個 class 的實體視爲第一個 class 的實體。

`compareTo()`契約的最後一段是個強烈建議而不是個真正條文，其中表示，以 `compareTo()`測驗相等性，結果應該與 `equals()`所得結果一致。如果遵守這項條文，我們稱以 `compareTo()`進行的排序行爲「與 *equals* 一致」（*consistent with equals*），否則稱爲「與 *equals* 不一致」（*inconsistent with equals*）。一個 class 即使具有「與 *equals* 不一致」的 `compareTo()`，還是可以工作，但是以此 class 爲元素的任何 sorted collections（有內在排序能力的群集類別），可能不再遵守相應之 collection interfaces（`Collection`、`Set` 或 `Map`）的通用契約。這是因爲這些 interfaces 的通用契約是根據 `equals()`定義的，而你的（上述的）sorted collections 卻採用 `compareTo()`代替 `equals()`進行相等性測試。發生這種情形並不一定是場災難，然而你應當對它有所認識。（譯註：C++ 標準程式庫亦存在相同的概念和實際情況，請參考《*Effective STL*》,by Scott Meyers, 條款 19：Understand the difference between equality and equivalence）

舉個例子，考慮 `BigDecimal` class，它的 `compareTo()`與 `equals()`不一致。如果你創建一個 `HashSet`，然後爲它添加新元素 `new BigDecimal(-0.0f)`和 `new BigDecimal(0.0f)`，這個 set 將含有 2 個元素，這是因爲以 `equals()`比較上述兩個 `BigDecimal` 元素，結果並不相等（譯註：因此得以都被放入「元素不得重複」的 set 中）。如果改用 `TreeSet` 而不是 `HashSet`，進行同樣過程，將只含有 1 個元素，因爲透過 `compareTo()`比較兩個 `BigDecimal`，結果相等（譯註：因此不得同時被放入「元素不得重複」的 set 中）（詳情請參考 `BigDecimal` 文件）。

撰寫 `compareTo()`和撰寫 `equals()`差不多，但有數個關鍵差異。你無需在型別轉換之前檢查引數型別。如果引數型別不適當，`compareTo()`應當拋出異常 `ClassCastException`。如果引數爲 `null`，`compareTo()`應當拋出異常 `NullPointerException`。這正是當你將引數轉換爲正確型別而後存取其欄位時，會得到的行爲。

「欄位比較」本身是「順序比較」而非「相等比較」。object reference 欄位的比較係以「遞迴呼叫 `compareTo()`」方式完成。如果某個欄位沒有實作 `compareTo()`，抑或你需要一個非標準排序，可使用一個明確的 *Comparator* 來代替。你可以自己寫一個，也可以使用預先存在者，例如**條款 7** 的 `CaseInsensitiveString` class 的 `compareTo()`那樣：

```
public int compareTo(Object o) {
    CaseInsensitiveString cis = (CaseInsensitiveString)o;
    return String.CASE_INSENSITIVE_ORDER.compare(s, cis.s);
}
```

比較基本型（primitive type）欄位，應當採用關係運算子 `<` 和 `>` ，比較 array，則應當對每個元素逐一施行這些原則。如果一個 class 具有多個有重要象徵意義的欄位，對它們進行比較時，先後次序頗爲重要。你應該先從最有意義（最重要）的欄位著手，以減少無謂工作。一旦比較得出非 0 值（0 代表相等），就結束比較，傳回結果。如果最具意義（最重要）的成員相等，再比較次具意義的欄位，依此類推。如果所有欄位都相等，則參與比較的兩個物件相等，傳回 0。下面以**條款 8** 的 `PhoneNumber` class 爲例，展示這項技術：

```
public int compareTo(Object o) {
    PhoneNumber pn = (PhoneNumber)o;

    // Compare area codes
    if (areaCode < pn.areaCode)
        return -1;
    if (areaCode > pn.areaCode)
        return  1;

    // Area codes are equal, compare exchanges
    if (exchange < pn.exchange)
        return -1;
    if (exchange > pn.exchange)
        return  1;

    // Area codes and exchanges are equal, compare extensions
    if (extension < pn.extension)
        return -1;
    if (extension > pn.extension)
        return  1;

    return 0;  // All fields are equal
}
```

雖然這個函式運作良好，但還有改進空間。還記得嗎，compareTo()契約之中並未把回返值的量（magnitude）列爲條件，只是規定回返值的正負號（sign）。你可以利用這一點簡化程式碼，或許可以使之運行得更快一些：

```
public int compareTo(Object o) {
    PhoneNumber pn = (PhoneNumber)o;

    // Compare area codes
    int areaCodeDiff = areaCode - pn.areaCode;
    if (areaCodeDiff != 0)
        return areaCodeDiff;

    // Area codes are equal, compare exchanges
    int exchangeDiff = exchange - pn.exchange;
    if (exchangeDiff != 0)
        return exchangeDiff;

    // Area codes and exchanges are equal, compare extensions
    return extension - pn.extension;
}
```

以上技巧在這兒運作良好，但我們應當戒慎恐懼。別輕易使用它，除非你確信我們所討論的欄位不可能爲負數，或更一般地說，你確信欄位值的下界和上界之間的差距小於或等於 Integer.MAX_VALUE(231-1)。這個技巧無法普遍適用的原因在於，一個帶正負號的 32-bit 整數，並沒有大到足以表示任意兩個「帶正負號 32-bit 整數」的差。如果 i 是個很大的正值 int，j 是一個很大的負值 int，(i-j)將溢位（overflow）並傳回負值。此後 compareTo()將無法正常工作，將對某些引數傳回荒謬的結果，因而違反 compareTo()契約的第一條文和第二條文。這並非只是純粹理論上的問題，這種情況已經在實際系統中引發故障。由於有問題的 compareTo()還是可以正常處理大多數資料，所以這一類故障很難排除。

4

類別和介面

Classes and Interfaces

Classes（類別）和 interfaces（介面）是 Java 語言的核心。它們是抽象性（abstraction）的基本單元。Java 語言提供了許多威力強大的元素用來設計 classes 和 interfaces。本章包含許多準則，可以協助你對這些元素做出最佳運用，使你的 classes 和 interfaces 更有用、強固、靈活。

條款 12：將 classes 和其成員的可存取性（accessibility）最小化

區分設計良好的模組和設計糟糕的模組，最重要的一個因素就是模組隱藏其內部資料和其他實作細目的程度。設計良好的模組會隱藏其所有實作細目，把 API 和實作細節乾淨地切割開來；模組和模組之間只透過 APIs 進行通訊，不需知道對方內部工作細節。這個概念稱爲資訊隱藏（information hiding）或封裝（encapsulation），是軟體設計的基本信條之一 [Parnas72]。

資訊隱藏的重要性是多方面的，大多數基於一個事實：它可以有效解除組成一個系統所需的模組和模組之間的耦合關係，允許它們獨立而各自地被開發、被測試、被最佳化、被使用、被理解、被修改。這可以加速系統的開發，因爲模組可以平行發展。它也緩和了維護上的負擔，因爲模組可以被快速理解和除錯，無需擔憂對其他模組造成傷害。雖然資訊隱藏本身並不會造成良好效率，但它能夠造成高實效性的效率調校（performance tuning）。一旦系統完成，而且評測（profiling）結果指出哪一個模組引發效率問題（**條款 37**），那些模組就可以被最佳化，不至於影響其他模組的正確性。資訊隱藏可以增加軟體的復用程度，因爲個別模組彼此並不互相依賴，因此往往在它們被開發出來的環境之外，仍然有用。最後一點，資訊隱藏減少了大型系統的構築風險，即使整個系統不成功，個別模組也有可能成功。

Java 語言有許多設施可以協助實現資訊隱藏。其中之一是存取控制（access control）機制 [JLS, 6.6]，用來決定 classes、interfaces 和 members 的可存取性。物體（entity）的可存取性由物體的宣告位置及宣告式中可能出現的存取飾詞（private、protected、public）決定。正確使用這些飾詞（modifiers）對於實現資訊隱藏是很有必要的。

經驗法則指出，你應該使每一個 class 或其成員儘可能不被外界存取。換句話說你在設計程式時應該根據軟體的功能，儘可能使用最低存取級別。

頂層（非巢狀的）classes 和 interfaces，只可能使用兩種存取級別：package-private 和 public。如果你對頂層 classes 和 interfaces 採用 public 飾詞，那麼它們就是 public，否則就是 package-private。如果一個頂層 classes 或 interfaces 實際運用上可以是 package-private，那就應該讓它成爲那樣的級別，於是它就成爲 package 實作品的一部份，而非 exported API 的一部分，於是你便可以在往後版本中修改它、替換它或撤銷它，不必擔心會傷害現有客戶。如果讓它成爲 public，你就被迫必須永遠支援它，以保證相容性。

如果 package-private 的頂層 classes 或 interfaces 僅在惟一一個 class 中被使用，你應該考慮使它成爲後者的一個巢狀的（nested）private class 或 interface（**條款 18**），這便更進一步降低其可存取性。然而這麼做的重要性還比不上「令一個非必要的 public class 成爲 package-private」（譯註：如上一段所述），因爲一個 package-private class 將成爲 package 實作品的一部分，而不是 API 的一部分。

成員（包括欄位、函式、巢狀類別、巢狀介面）有四種可能的存取級別，以下按存取性漸強的次序列出：

- private — 這種成員僅在其宣告式所在之頂層 class 內部可存取。
- package-private — 這種成員可被其宣告式所在之 package 內的任何 classes 存取。技術上稱之爲 *default*（預設）存取級別。如果你沒有爲成員指定任何存取飾詞，它就成爲這種級別。
- protected — 這種成員可被其宣告式所在之 class 的 subclasses 存取（但有一些限制 [JLS, 6.6.2]），也可被其宣告式所在之 package 內的所有 classes 存取。
- public — 可不受任何限制地被存取。

謹慎設計 class 的 public API 之後，下一步工作應該是使所有其他成員變成 private。只有當同一個 package 內的其他 classes 確實需要存取某個成員時，才將其 private 飾詞移除，使它成爲 package-private。如果你發現自己竟然頻繁地做這項工作，那就應該重新檢討設計，看看是否能分解出新的 classes，以減少耦合。private 成員和 package-private 成員都是 class 實作碼的一部份，一般不會影響其 exported API。但如果 class 實現了 `Serializable`（**條款 54**、**條款 55**），這些欄位就有可能洩漏（leak）到 exported API 中。

當 public classes 的成員的存取級別從 package-private 變成 protected 時，其可存取性會大幅增加。protected 成員是 class exported API 的一部份，必須永遠獲得支援。此外，一個 exported class 的 protected 成員對於 class 內部實作碼而言，相當於做出了等同於 public 的承諾（**條款 15**）。protected 成員的需求應該是相當稀少的。

有一條規則限制了你「降低函式可存取性」的能力：如果某個函式重載了 superclass 中的函式，那麼它在 subclass 中的存取級別不得低於它在 superclass 中的存取級別 [JLS, 8.4.6.3]。這條規則是必要的，可確保「superclass 實體可用之處，subclass 實體也一定可用」。如果違背這條原則，當你試圖編譯 subclass 時，編譯器會發出錯誤訊息。這條規則的一個特殊情況是，如果某 class 實現了一個 interface，那麼 class 之中相應於 interface 的所有函式都必須宣告爲 public，這是因爲 interface 的所有函式都暗自爲 public。

Public classes 基本上很少擁有 public 欄位（但常常擁有 public 函式）。因爲如果欄位是個 nonfinal，或是個指向某可變物件（*mutable* object）的 final reference，那麼一旦它成爲 public，你就喪失了對於「儲存於此欄位中的值」的約束能力；你也將喪失「欄位被修改時」的任何應變能力。一個簡單的後果就是：具有 public *mutable* 欄位的 class 在多緒環境下並不安全（亦即 not thread-safe）。即使欄位是 final，而且不是個「指向（*refer*）*mutable* 物件」的 reference，如果其存取級別是 public，你也將喪失彈性，不復能夠把該欄位轉換（調動，*switch*）至某個嶄新內部資料表述（其中並不存在該欄位）內。

「public classes 不該擁有 public 欄位」這條規則存在一個例外：public classes 可透過 public static final 欄位來表現常數。習慣上這種欄位的名稱以大寫字母開頭，再以底線分隔文字（**條款 38**）。這些欄位應該只包含基本型數值，或包含「指向不可變物件」的 reference（**條款 13**）。final 欄位如果內含「指向可變物件」之 reference，將會沾惹 nonfinal 欄位的所有缺點。是的，儘管其中的 (final) reference 不能被修

改，但 (final) reference 所指向的「被引用物件」可被修改，這會帶來災難。

注意：長度不爲 0 的 array 總是可變的（*mutable*），所以 class 之中如果出現 public static final array 欄位，幾乎總是錯誤的。如果 class 之中存在這樣的欄位，客戶便得以修改 array 內容，而這往往成爲安全漏洞（security holes）的根源：

```
// Potential security hole!
public static final Type[] VALUES = { ... };
```

這樣的 public array 應該被替換爲一個 private array 和一個 public *immutable* list：

```
private static final Type[] PRIVATE_VALUES = { ... };

public static final List VALUES =
   Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

如果你需要編譯期型別安全性（compile-time type safety），並且願意損失一些效率的話，也可以使用一個 public 函式傳回 private array 副本，以此取代 public array 欄位：

```
private static final Type[] PRIVATE_VALUES = { ... };

public static final Type[] values() {
   return (Type[]) PRIVATE_VALUES.clone();
}
```

總而言之，你應該儘可能降低可存取性。在謹慎設計了一個最小程度的 public API 之後，應該防止任何離題歧路的 classes, interfaces 或 members 成爲 API 的一部分。除了前述的 public static final 欄位例外情況，public classes 不應該擁有 public 欄位。而在前述例外情況中，請確保被 public static final 欄位所引用的物件是不可變的（*immutable*）。

條款 13：偏愛不變性（immutability）

所謂 *immutable* class（不可變類別）就是其實體不能被修改的 class。包含在這種 class 的實體內的所有資訊都在創建之初提供，並在物件生存期間固定不變。Java 標準程式庫包含許多 *immutable* class，例如 `String`、基本外覆類別（primitive wrapper classes）、`BigInteger` 和 `BigDecimal`。這種 classes 的存在有很多好理由：和 *mutable* classes（可變類別）相比，它更容易設計、實現和使用，不易犯錯而且更安全。

製造一個 *immutable* class（不可變類別）時，請遵循以下 5 個原則：

- 不要提供任何「可修改物件內容」的函式（這類函式稱爲改動式，*mutator*）。
- 保證沒有任何函式可被覆寫（overridden）。這可防止粗心或惡意的 subclasses 危害物件的「不可變性」。防止函式被覆寫的一般作法是使 class 成爲 final，但還有其他選擇，稍後介紹。
- 令所有欄位爲 final。借助語言系統的強制性，清楚表達意圖。而且，在不使用同步機制（synchronization）的情況下，如果新建物件的 reference 從一個執行緒被傳遞到另一個執行緒，爲確保「正確的行爲」（所謂正確的行爲取決於爲修訂記憶體模型而做的努力的結果 [Pugh01a]），這一條也是必需的。
- 令所有欄位都是 private。這可以防止客戶直接修改欄位。儘管技術上而言 *immutable* classes 可以擁有 public final 欄位，內含基本型別或「指向 *immutable* 物件」的 reference，但實際上這種作法並不受到推薦，因爲這會造成無法改動日後新版本的內部表述（**條款 12**）。
- 確保對任何可變組件（*mutable* components）的互斥存取（exclusive access）。如果你的 class 中有任何欄位指向可變物件（*mutable* objects），你必須確保其客戶不能獲得這些物件的 references。千萬不要以客戶提供的 object reference 來初始化這樣的欄位，也不要從存取式（accessor）中傳回 object reference。請在建構式、存取式和 `readObject()`（**條款 56**）中使用保護性拷貝（defensive copy，**條款 24**）。

先前各條款中有許多 classes 是不可變的（*immutable*）。**條款 8** 的 `PhoneNumber` 就是一例，它對每個屬性（attribute）提供了一個 accessor（存取式），但沒有相應的 mutator（改動式）。下面是一個稍微複雜的例子。

```
public final class Complex {
    private final float re;
    private final float im;

    public Complex(float re, float im) {
        this.re = re;
        this.im = im;
    }

    // Accessors with no corresponding mutators
    public float realPart()     { return re; }
    public float imaginaryPart() { return im; }

    public Complex add(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }

    public Complex subtract(Complex c) {
        return new Complex(re - c.re, im - c.im);
    }

    public Complex multiply(Complex c) {
        return new Complex(re*c.re - im*c.im,
                           re*c.im + im*c.re);
    }

    public Complex divide(Complex c) {
        float tmp = c.re*c.re + c.im*c.im;
        return new Complex((re*c.re + im*c.im)/tmp,
                           (im*c.re - re*c.im)/tmp);
    }

    public boolean equals(Object o) {
       if (o == this)
          return true;
       if (!(o instanceof Complex))
          return false;
       Complex c = (Complex)o;
       return (Float.floatToIntBits(re) ==    // See page 33 to
               Float.floatToIntBits(c.re)) && // find out why
              (Float.floatToIntBits(im) ==    // floatToIntBits
               Float.floatToIntBits(c.im));   // is used.
    }
```

```
    public int hashCode() {
        int result = 17 + Float.floatToIntBits(re);
        result = 37*result + Float.floatToIntBits(im);
        return result;
    }

    public String toString() {
        return "(" + re + " + " + im + "i)";
    }
}
```

這個 class 用來表現複數（complex number，具有實部和虛部）。除了標準的 `Object` 函式，它還提供了實部和虛部存取式，以及四種基本算術操作：加、減、乘、除。注意，這些算術操作都創建並傳回一個新的 `Complex` 實體，而不是修改原實體內容。這種作法被用於一些重要的 *immutable* classes 中，稱爲 ***functional approach***，因爲它傳回「對運算元施行某個函式（function）」而產生的結果。對比於此法但更常見的是 ***procedural approach***：對運算元施行某個程序（procedure），造成運算元的狀態改變。（譯註：此處 function 和 procedure 的意義區隔是：不改變運算元內容的，稱爲 function；會改變運算元內容的，稱爲 procedure）

如果你不熟悉 ***functional approach***，可能覺得它不很自然，但它確實實現了「不可變性」（*immutability*），後者擁有許多優點。首先，*immutable* objects 比較簡單，僅存在一種狀態，即它當初被創建時的狀態。如果你能確定所有建構式建立起 class 約束條件（invariants），那些條件將永保不變，不需要你或 classes 用戶做任何額外工作。至於 *mutable* objects 則有任意複雜的狀態空間。如果文件中沒有對改動式（mutator）進行的狀態轉移（state transitions）提供精確描述，那麼全心信賴地運用一個 *mutable* class 將是困難甚至不大可能的。

Immutable objects 本質上是多緒安全的（thread-safe），不需同步機制的輔助。它們不會因爲多緒並行存取（accessing concurrently）而遭受破壞。這是獲得多緒安全性的最簡單途徑。事實上，沒有任何執行緒會看到其他執行緒對 *immutable* object 的影響。因此 *immutable* object 可以被自由地共享。*immutable* classes 應該鼓勵客戶儘可能復用其既有實體。達到這一點的一個簡單辦法就是，爲頻繁使用的數值提供一個對應的 public static final 常數。例如 `Complex` class 可以提供以下常數：

```
public static final Complex ZERO = new Complex(0, 0);
public static final Complex ONE  = new Complex(1, 0);
public static final Complex I  =   new Complex(0, 1);
```

更進一步的作法是讓 *immutable* class 提供一個 static ***factories***，將經常被索求的實體以快取（cache）方式儲存起來，避免在「被索求的實體」存在時重複創建實體。`BigInteger` 和 `Boolean` classes 都具有這種 static ***factories***。藉由這種設計，客戶可以共享既有實體，不必創建新實體，可以降低記憶體用量及垃圾回收成本。

immutable objects 可被自由共享，這使得我們永遠不必對它使用保護性拷貝（defensive copies，**條款 24**）。事實上根本不必做任何拷貝，因爲任何拷貝都與初始物件等價，因此不必要也不應該對 *immutable* class 提供 `clone()`和 *copy* 建構式（**條款 10**）。這一點在早期的 Java 不易被理解，所以 `String` class 還是有個 *copy* 建構式，但它應該很少被使用（**條款 4**）。

你不僅可以共享 *immutable* objects，還可以共享它們的內部資訊。例如 `BigInteger` 內部使用一個帶正負號數值，其中的正負號以 `int` 表示，數值以 `int` array 表示。`negate()`用來產生一個數值相同而正負號相反的 `BigInteger` 物件，它不需要拷貝上述的 array，只要將新建的 `BigInteger` 指向初始實體的內部 array 即可。

Immutable objects 對其他物件（無論是可變或不可變的）形成了大量積木（構件）。如果我們知道構成複雜物件的組件物件（component objects）是不可變的，那麼維護複雜物件的不變性會比較容易些。這一原則的特殊例子是：以 *immutable* objects 製造出大量的 map 鍵值和 set 元素；一旦它們進入 map 或 set，你其實無需擔心其值會被改變，因爲如果其值發生變化，會破壞 map 或 set 的約束條件。

immutable classes 真正而且惟一的缺點是，每個不同的值需要一個獨立物件。創建這些物件可能帶來不小的成本，特別是面對大型物件。假設你有一個百萬位元（million-bit）的 `BigInteger`，你需要對其低位元取補數：

```
BigInteger moby = ...;
moby = moby.flipBit(0);
```

`flipBit()`會創建出一個新的 `BigInteger` 實體，同樣是百萬位元，但與原實體僅有一個位元的差別。此一操作所需的時間和空間，與 `BigInteger` 的尺寸呈比例。但是 `java.util.BitSet` 就不一樣。它和 `BigInteger` 同樣用來表現一個任意長度的位元序列，但卻是可變的。`BitSet` class 提供一個函式，允許你在常數時間內改變實體中的百萬位元內的任一位元狀態。

如果你執行一個多步操作（multistep operation），每一步產生一個新物件，最終只保留最後結果，那麼效率問題就突顯出來了。有兩個辦法可以解決這一問題，第一個辦法是假想哪些多步操作會被普遍需要，並將它們視同基本單元來提供。如果一個多步操作被當做基本單元提供出來，*immutable* class 就不必每一步創建一個獨立物件。*immutable* class 的內部作法可以非常靈活，例如 `BigInteger` 具有一個 package-private *mutable* companion class（可變的配套類別），用來加速諸如 modular exponentiation（模數取冪）等多步運算。由於先前提到的諸多原因，使用 *mutable* companion class 十分困難，幸運的是 `BigInteger` 爲你做掉了所有困難工作。

如果你能夠準確預測客戶想要在你的 *immutable* class 上做哪些複雜的多步操作，那麼上述辦法可以運行良好。否則最好是提供一個 public *mutable* companion class。這個辦法的最有名例子是 Java 標準程式庫中的 `String`，它的 *mutable* companion（可變的配套類別）是 `StringBuffer`。我們也可以說，`BitSet` 在某些情況下扮演了 `BigInteger` 的 *mutable* companion 的角色。

現在，你已經知道如何實現一個 *immutable* class，也理解了「不可變性」的優缺點，我們再來討論另一些設計方案。爲了保證「不可變性」，必須使 class 的任何函式不得被覆寫（overridden）。讓 class 成爲 final 當然可以保證這一點，此外另有兩個辦法。其一是使 class 中的所有函式（而非 class 本身）成爲 final。這種作法的惟一好處是程式員得以擴展 class — 在舊 class 身上添加新函式。這種作法與「在個別的、不可實體化的 utility class（**條款 3**）身上提供新的靜態函式」效果相同，因此不推薦使用。

第二個方案是，令 class 的所有建構式爲 private 或 package-private，然後添加 public static ***factories***，代替 public 建構式（**條款 1**）。爲了更具體說明，我以 `Complex` 爲例：

```
// Immutable class with static factories instead of constructors
public class Complex {
    private final float re;
    private final float im;

    Complex(float re, float im) {
        this.re = re;
        this.im = im;
    }

    public static Complex valueOf(float re, float im) {
        return new Complex(re, im);
    }

    ... // Remainder unchanged
}
```

儘管這一方案並沒有被普遍使用，卻往往是三種方案中最好的一種選擇。它最有彈性，因爲它允許使用多個 package-private implementation classes。對於 package 以外的客戶，*immutable* class 實際上本來就是 final，因爲要擴展一個「來自其他 package 而又缺乏 public 或 protected 建構式」的 class 是不可能的。除了這種彈性外，這一方案也使得「在日後新版本中藉由提高 static ***factories*** 的 object-caching 能力來調校程式效率」成爲可能。

條款 1 討論過，static ***factories*** 和建構式相比，有許多好處。例如，假設 `Complex` 有必要提供「根據極坐標創建一個複數」的函式。使用建構式會很麻煩，因爲所有建構式都具有相同署名式：`Complex(float,float)`。改用 static ***factories*** 會使問題變得比較簡單：只要添加第二個 static ***factory***，並爲它取一個「足可表達其功能」的函式名稱即可：

```
public static Complex valueOfPolar(float r, float theta) {
    return new Complex((float) (r * Math.cos(theta)),
                       (float) (r * Math.sin(theta)));
}
```

當初設計 `BigInteger` 和 `BigDecimal` 時，「*immutable* classes 實際上即爲 final」這一點尚未被大眾廣泛理解，因此這兩個 classes 的所有函式都可被覆寫。不幸的

是，爲了保持回溯相容，這個問題無法修正。因此如果你寫了一個 class，其安全性（security）取決於 `BigInteger` 引數或 `BigDecimal` 引數（由不可信賴之客戶提供）的不可變性，那麼你必須檢查看看引數是否確實是個 `BigInteger` 實體或 `BigDecimal` 實體，而不是一個不可信賴的 subclass 實體。如果是後面這種情況，就必須在「引數也許可變」的假設下進行「保護性拷貝，defensively copy」（**條款 24**）：

```
public void foo(BigInteger b) {
    if (b.getClass() != BigInteger.class)
        b = new BigInteger(b.toByteArray());
    ...
}
```

本條款一開始有關於 *immutable* classes 的一系列規則指出，不允許存在任何函式得以修改 *immutable* classes 的物件內容，而且所有欄位都應該是 final。事實上這些規則有點過於嚴格。爲提高效率，可以放鬆限制。實際使用中並無哪個函式可以對物件狀態產生可見的變化。然而許多 *immutable* classes 會有一個或多個 nonfinal 贅餘欄位，並以快取（cache）方式在其中儲存昂貴計算所得的第一次運算結果。如果未來又有相同的計算需求，就傳回被快取儲存下來的物件，節省再次計算成本。這種技巧之所以能夠有效運作，因爲物件是不可變的，因此一旦重新執行計算，保證得到相同結果。

舉個例子，`PhoneNumber`（**條款 8**, p.40）的 `hashCode()`第一次被呼叫時，會計算 hash 碼，並將結果存起來以應付將來的再次需要。這種技術 — 典型的惰式初始化手法（lazy initialization，**條款 48**）— 也用於 `String` class。它不需要同步機制，因爲即使 hash 碼被重複計算也不會產生問題。下面是 *immutable* object 的惰性初始化函式（傳回一個快取值）的一般用法：

```
// Cached, lazily initialized function of an immutable object
private volatile Foo cachedFooVal = UNLIKELY_FOO_VALUE;

public Foo foo() {
    Foo result = cachedFooVal;
    if (result == UNLIKELY_FOO_VALUE)
        result = cachedFooVal = fooValue();
    return result;
}

// Private helper function to calculate our foo value
private Foo fooVal() { ... }
```

這裡應該對 serializability（序列化能力）提出警告。如果你的 *immutable* class 需要實現 `Serializable`，而 class 之中又包含一或多個欄位指向 *mutable* objects，那麼你就必須明確提供一個 `readObject()`或 `readResolve()` — 即使預設的序列化格式（serialized form）是你可以接受的。預設的 `readObject()`允許攻擊者從你的 *immutable* class 創建出一個 *mutable* 實體。這個問題將在**條款 56** 詳細討論。

總而言之，請不要爲每一個取值函式（getter）都提供一個相應的設值函式（setter）。你應該儘量將 class 設計是不可變，除非有很好的理由才讓它成爲可變。*Immutable* classes 有許多優點，惟一缺點是某些情況下潛在著效率問題。你應該總是讓小型數值物件（small value objects）不可變，例如 `PhoneNumber` 和 `Complex`。Java 標準程式庫中有一些 classes 應該是不可變的，例如 `java.util.Date` 和 `java.awt.Point`，當時卻沒有那麼設計。你也應該認真考慮讓較大型數值物件（larger value objects）例如 `String` 和 `BigInteger` 成爲不可變。只有在確實需要考慮效率因素時（**條款 37**），才考慮爲你的 *immutable* class 提供一個 public *mutable* companion class（可變的配套類別）。

對某些 classes 而言，「不可變性」其實並不實際，例如對於 "process classes" 如 `Thread` 和 `TimerTask`。但如果某個 class 無法成爲不可變，你還是應該儘可能限制其可變性。減少物件存在所需的狀態個數（number of states），可以更輕鬆地分析物件，降低出錯的可能。所以建構式應該創建出完全初始化的物件，建立起所有約束條件（invariants）。建構式不該向其他函式傳遞半成品實體。除非有絕對充份的理由，否則不應該在建構式之外另提供 public 初始化函式。同樣道理，你不應該提供一個「再初始化」函式，它會使物件得以再度被使用，猶如「以另一個初始狀態重新被建構」。「再初始化」函式帶來複雜度的增加，相較之下，它並沒有帶來什麼效率優勢。

`TimerTask` class 是以上原則的良好實例。它可變（*mutable*），但狀態空間故意設計得很小。你爲它創建一個實體後，可以將它納入排程，也可以取消它。一旦某個計時任務（timer task）結束，或是被取消，你將不能夠對它再次排程。

最後還請注意一點，本條款的 `Complex` class 僅用來說明不變性（immutability），而不是一個具備工業強度的實作品。`Complex` 對複數的乘除使用了標準公式，但這樣會造成不正確的四捨五入，而且對複數的 NaNs 和無窮值（infinities）沒有提供良好的語意表述 [Kahan91, Smith62, Thomas94]。

條款 14：優先考慮複合（composition），然後才是繼承（inheritance）

繼承是實現「程式碼復用」的一條有力途徑，但它並非總是最佳選擇。如果不適當地使用它，會導致脆弱的軟體。在 package 之內使用繼承是安全的，因爲這時候 subclass 和 superclass 的實作碼在同一個程式員的控制下。對於專門設計用來繼承，並有詳細擴展說明文件的 classes（**條款 15**），使用繼承也是安全的。然而，跨越 package 邊界，對普通的、具象的 class 實施繼承則是危險的。請注意，本書以繼承（inheritance）一詞代表「實作繼承」（implementation inheritance），也就是以 class 擴展另一個 class。本條款討論的問題並不適用於介面繼承（interface inheritance），也就是以 class 實現一個 interface，或以 interface 擴展另一個 interface。

和函式呼叫有所不同，繼承打破了封裝性（encapsulation）[Snyder86]。換句話說 subclass 本身的功能和其 superclass 的實作細節息息相關。superclass 的實作細節有可能隨版本而變異，果真如此，即使 subclass 的程式碼不被觸動，subclass 也會受到破壞。結果，subclass 必然與 superclass 像雙輪馬車似地緊緊關聯在一起，除非 superclass 是「專爲被擴展（extended）」而設計，並帶良好的說明文件。

爲了更具體些，讓我們假設有一個程式使用 `HashSet`。爲了改善程式效率，我們需要知道自 `HashSet` 被創建之後曾經添加過多少個元素（不要和其當前大小混淆，大小會隨著元素的移除而下降）。爲了提供這種功能，我們寫了一個 `HashSet` 變體，在其中加入一個變數，用以記錄被安插的元素總數，並爲該變數提供一個存取式（accessor）。`HashSet` class 內含兩個函式可以添加元素：`add()`和 `addAll()`，所以我們需要覆寫這兩個函式：

```
// Broken - Inappropriate use of inheritance!
public class InstrumentedHashSet extends HashSet {
    // The number of attempted element insertions
    private int addCount = 0;

    public InstrumentedHashSet() {
    }

    public InstrumentedHashSet(Collection c) {
        super(c);
    }
```

```
    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor);
    }

    public boolean add(Object o) {
        addCount++;
        return super.add(o);
    }

    public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }
}
```

這個 class 看起來合理，但無法有效運作。假設我們爲它創建一個實體，然後以 `addAll()`添加三個元素：

```
InstrumentedHashSet s = new InstrumentedHashSet();
s.addAll(Arrays.asList(new String[] {"Snap","Crackle","Pop"}));
```

此時我們希望 `getAddCount()`傳回 3，但它實際上卻傳回 6。哪裡出錯了？原來，在內部實作中，`HashSet` 的 `addAll()`是以其 `add()`爲基礎實作出來的，但 `HashSet` 的文件沒有說明這一細節（這並不奇怪）。`InstrumentedHashSet` 的 `addAll()` 在 `addCount` 上增加 3，然後透過 `super.addAll()`呼叫 `HashSet` 的 `addAll()`。這會導致喚起 `InstrumentedHashSet` 所覆寫的那個 `add()` — 每個元素呼叫一次。三次呼叫分別爲 `addCount` 各加 1，因此共增加 6：因爲 `addAll()`所添加的每一個元素都被計數兩次。

我們可以修正 subclass，作法是取消它對 `addAll()`的覆寫。然而儘管如此獲得的 class 可以有效運作，其正確功能卻取決於一個事實：`HashSet` 的 `addAll()`奠基於 `add()`之上。這種「自用性（self-use）」只是某種實作細節，不會被 Java 平台上的所有實作品承諾遵守，而且可能隨版本而異。因此，這樣而獲得的 `InstrumentedHashSet` class 是脆弱的。

稍微好一點的改良辦法是覆寫 `addAll()`，讓它在指定的群集（collection）身上進行迭代（iteration），並對每一個元素呼叫 `add()`。這保證獲得正確結果，不論 `HashSet` 的 `addAll()`是否奠基於 `add()`之上，因爲此時 `HashSet` 的 `addAll()`不會再次被

喚起。然而這種技術並沒有解決我們的所有問題，它實際上等於重新實作 superclass 的函式（也許「自用」，也許不是），這不但困難、耗時，也容易出錯，而且並非總是可行，因爲某些函式必須存取 private 欄位，而 private 欄位在 subclass 內並不具可存取性。

造成 subclasses 脆弱的一個原因是，其 superclass 可以取得後續版本中的新函式。假設有個程式，其安全性取決於一個事實：所有被安插到某個群集內的元素都必須滿足某種條件（predicate）。此條件可經由以下措施獲得保證：撰寫該群集的 subclass，覆寫所有「有能力向群集添加元素」的函式，保證元素被添加至群集前必須先滿足某條件。這個辦法可行的前提是：後繼版本不再爲 superclass 增加「可添加元素」的新函式。但如果 superclass 添加了上述新函式，那麼呼叫新增函式（未被 subclass 覆寫）就可能在 subclass 實體中添加一個「非法」元素。這不僅是單純理論上的問題。當我翻新 `Hashtable` 和 `Vector` 以求融入 Collections Framework 時，就修正了數個這一類安全漏洞（security holes）。

上面所有問題都源於函式的覆寫。你或許會認爲，擴展一個 class 時，只增加新函式而不覆寫現有函式，必然就是安全的。雖然這種擴展方式的確比較安全，但也不是毫無風險。如果後繼版本中 superclass 需要新函式，而你很不幸地已經給予 subclass 一個函式，有著相同的署名（signature）但不同的回返值型別，那麼你的 subclass 將無法通過編譯 [JLS, 8.4.6.3]。如果你已經給予 subclass 一個函式，其署名及回返值型別與 superclass 的新函式相同，那麼事實上你便是在對它進行覆寫，於是你會遭遇稍早描述的兩個問題。此外，subclass 函式是否會履行 superclass 的新函式的契約（contract），不無疑問，因爲當你撰寫 subclass 函式時，該契約尚未被寫出。

幸運的是，有一種辦法可以避免先前提到的所有問題。我們不要擴展現有的 class，而是爲新 class 添加一個 private 欄位，用來指向現有 class 的一份實體。這種設計稱爲複合（**composition**），因爲現有的 class 成了新 class 的一部分。新 class 中的每一個 *instance* 函式都呼叫其內所含之「既有 class 的實體」的相應函式，並傳回其結果。這便是所謂的轉發（forwarding），而新 class 中的函式稱爲轉發函式（forwarding method）。這樣所產生的 class 便是穩定的，並不依存於既有 class 的

實作細節。這時候即使對既有 class 添加新函式，也不會影響新 class。爲了更具體說明，下面是先前 `InstrumentedHashSet` 的一個替代版本，使用「複合/轉發」（composition/forwarding）手法：

```
// Wrapper class - uses composition in place of inheritance
public class InstrumentedSet implements Set {
    private final Set s;
    private int addCount = 0;

    public InstrumentedSet(Set s) {
        this.s = s;
    }

    public boolean add(Object o) {
        addCount++;
        return s.add(o);
    }

    public boolean addAll(Collection c) {
        addCount += c.size();
        return s.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }

    // Forwarding methods
    public void clear()                 { s.clear();           }
    public boolean contains(Object o)   { return s.contains(o); }
    public boolean isEmpty()            { return s.isEmpty();  }
    public int size()                   { return s.size();     }
    public Iterator iterator()          { return s.iterator(); }
    public boolean remove(Object o)     { return s.remove(o);  }
    public boolean containsAll(Collection c)
                                        { return s.containsAll(c); }
    public boolean removeAll(Collection c)
                                        { return s.removeAll(c);   }
    public boolean retainAll(Collection c)
                                        { return s.retainAll(c);   }
    public Object[] toArray()           { return s.toArray();  }
    public Object[] toArray(Object[] a) { return s.toArray(a); }
    public boolean equals(Object o)     { return s.equals(o);  }
    public int hashCode()               { return s.hashCode(); }
    public String toString()            { return s.toString(); }
}
```

`InstrumentedSet` 的設計因 `Set` interface 的存在而可行，後者獲得了 `HashSet` 的功能。除了變得更強固，這樣的設計也非常有彈性。`InstrumentedSet` 實現 `Set` interface，擁有惟一建構式，其引數型別也是 `Set`。本質上，這個 class 把一種 `Set` 型別轉化爲另一種，並添加了儀表（instrumentation。譯註：意指計數器）功能。前面所提的繼承只在單一 concrete class 情況下有用，而且對 superclass 支持的每一個建構式必須有個別的建構式對應之，但現在這個外覆類別（wrapper class）可用來包裝任何 `Set` 實作品，可以和任何既有建構式一起工作。例如：

```
Set s1 = new InstrumentedSet(new TreeSet(list));
Set s2 = new InstrumentedSet(new HashSet(capacity, loadFactor));
```

`InstrumentedSet` class 甚至可被用來暫時取代一個並沒有儀表功能的 set 實體：

```
static void f(Set s) {
    InstrumentedSet sInst = new InstrumentedSet(s);
    ... // Within this method use sInst instead of s
}
```

每個 `InstrumentedSet` 實體都包覆一個 `Set` 實體，所以 `InstrumentedSet` 又被稱爲 wrapper class（外覆類別），或被稱爲 ***Decorator***（修飾器）範式 [Gamma95, p175]，這是因爲 `InstrumentedSet` 因爲添加儀表而「修飾了」set。有時候「複合（composition）與轉發（forwarding）」的組合被誤以爲是「委託」（delegation）。就技術而言，這不是委託，除非 wrapper object（外覆物件）把自己傳遞給 wrapped object（被覆物件）[Gamma95, p20]。

wrapper classes（外覆類別）的缺點很少。需要注意的是，wrapper classes 不適合用於回呼框架（callback frameworks） — 這種環境下物件把自己傳遞給其他物件做爲日後回呼（callback）之用。由於 wrapped object（被覆物件）並不知道其 wrapper，所以它傳出一個「指向自己」的 reference（亦即 `this`），回呼時繞離了 wrapper。這便是所謂的 SELF 問題 [Lieberman86]。有人擔心轉發函式（forwarding methods）會造成效率影響，或擔心 wrapper objects 帶來記憶體用量影響。事實證明這兩類問題都沒有太大衝擊。編寫轉發函式是有些單調，但由於你只需寫一個建構式，單調也就獲得了一些補償。

只有當 subclass 確實是 superclass 的一個子型別（subtype）時，才適合使用繼承。換句話說，對 classes A 和 B 而言，如果「B **是一種** A」的關係存在，B 才應該擴展 A。當你企圖以 B 擴展 A 時，問自己這樣一個問題：「每個 B 都是一個 A 嗎？」如果答案是否定的，那麼通常應該把 A 當做 B 的一份 private 實體，然後顯露更小、更簡單的 API：A 不是 B 的基本部分，只是其實作細節。

在 Java 標準程式庫中，許多地方明顯違背這條原則。例如 stack 不是一個 vector，所以 `Strack` 不該擴展 `Vector`。同樣道理，property list 不是 hash table，所以 `Properties` 不該擴展 `Hashtable`。上述兩種情況使用複合（composition）更爲合適。

如果在適合採用「複合」之處，你卻運用「繼承」，會非必要地暴露實作細節。導致 API 過份依賴原始實作細節，永遠限制你的 class 效率。更嚴重的是，如果暴露內部細節，客戶會直接存取它們。極端情況下這會導致語意混淆。例如 `p` 指向一個 `Properties` 實體，那麼 `p.getProperty(key)`可能會與 `p.get(key)`產生不同結果：前者使用 `Properties` 預設值，後者從 `Hashtable` 繼承而來，因此不這樣做。最壞情況下客戶可能會因爲直接修改 superclass 而破壞了 subclass 的約束條件（invariants）。在 `Properties` 裡頭，設計者希望只有字串可以作爲鍵值（key）和實值（value）。但若直接存取底層的 `Hashtable`，可能導致這一約束遭受破壞。一旦如此，我們就不再可能使用 `Properties` API 的其他部分（例如 `load` 和 `store`）。等到發現這個問題，已經太晚而無法更正了，因爲客戶將倚賴非字串的鍵值（key）和實值（value）。

在繼承與組合的抉擇中，當你決定使用前者前，還應該問自己最後一組問題：你試圖擴展的那個 class，其 API 還有任何缺陷嗎？如果是，你能夠容忍它們傳播給你的 class API 嗎？「繼承」將會傳播 superclass API 的所有缺陷，而「複合」允許你設計新的 API，隱藏 superclass 的缺陷。

總而言之，繼承的功能是強大的。但此同時，由於它違背了封裝性，因而也帶來了問題。只有當 subclass 和 superclass 之間存在確實的 subtype 關係，才適合運用繼承。即使如此，如果 subclass 和 superclass 存在於不同的 package 中，而且 superclass 並非爲了將來的擴展而設計，那麼「繼承」也會造成程式的脆弱性。爲避免這種脆弱性，應該使用複合和轉發（composition + forwarding）代替繼承（inheritance），

尤其如果存在一個合適的 interface 來實現 wrapper class — 因爲 wrapper classes 不只比 subclasses 更強固，功能也更強大。

條款 15：除非專為繼承而設計並提供文件，否則不要使用繼承

條款 14 提醒你，對一個並非專爲繼承而設計，而且也沒有完善文件的「外來 class」進行 *subclassing*（子類別化）是危險的。那麼，什麼是「專爲繼承而設計」？什麼又是完善的文件？

首先，這樣的 class 必須提供文件，準確描述覆寫（*overriding*）任何函式的效果。換句話說 class 必須說明它對於可被覆寫函式（overridable methods）的自用（self-use）程度：文件中必須說明，每一個 public 或 protected 函式或建構式，呼叫了哪一個可被覆寫函式，以什麼順序呼叫。每個呼叫所得結果將會如何影響後繼的處理（所謂「可被覆寫」，意味 nonfinal 而且 public 或 protected）。更一般地說，文件中必須指出 class 在什麼環境（情況）下可以呼叫一個可被覆寫函式。例如呼叫動作也許來自背景執行緒或靜態初始式（static initializers）。

按慣例，任何函式如果呼叫了可被覆寫函式，應該在其文件末尾描述這些呼叫動作。描述文字應該以「本實作碼」（"This implementation"）起頭。這樣的句子並非暗示 class 的行爲會隨著版本而變動，它只是暗示此一描述與函式內部細節有關。下面這個示例拷貝自 `java.util.AbstractCollection` 規格書（譯註：由於此處是爲了示範 Java 文件的模樣，所以保持原樣不譯）：

`public boolean remove(Object o)`

> Removes a single instance of the specified element from this collection, if it is present (optional operation). More formally, removes an element `e` such that `(o==null ? e==null : o.equals(e))`, if the collection contains one or more such elements. Returns `true` if the collection contained the specified element (or equivalently, if the collection changed as a result of the call).
>
> **This implementation** iterates over the collection looking for the specified element. If it finds the element, it removes the element from the collection using the iterator's `remove` method. Note that this implementation throws an `UnsupportedOperationException` if the iterator returned by this collection's `iterator` method does not implement the `remove` method.

這份文件清楚說明了覆寫 `iterator()`函式將會影響 `remove()`函式。此外它也確切描述了被 `iterator()`傳回之 `Iterator`，其行爲將會如何影響 `remove()`的行爲。對比於此的是**條款 14**，其中程式員只是簡單地將 `HashSet` 子類別化（*subclassing*），卻沒有說明覆寫 `add()`是否會影響 `addAll()`。

但是，這豈不違背了「優越的 API 文件應該描述函式做了什麼，而不是描述函式如何做」的格言嗎？確實如此！這是「繼承（inheritance）違反封裝性（encapsulation）」這一事實的不幸後果。爲了詳細說明一個 class，使它得以被安全地 *subclassed*（子類別化），你必須描述其實作細節。

爲了繼承而進行的設計，不僅只是加上自用（self-use）形式的詳細文件而已。爲了讓程式員寫出高效的 subclasses 而不必承受太多痛苦，class 必須以審慎挑選出來的 protected 函式（或 protected 欄位，但很罕見）提供內部活動的鉤子（hook）。以 `java.util.AbstracList` 的 `removeRange()`爲例（譯註：如前，保持原樣不譯）：

```
protected void removeRange(int fromIndex, int toIndex)
```

> Removes from this list all of the elements whose index is between `fromIndex`, inclusive, and `toIndex`, exclusive. Shifts any succeeding elements to the left (reduces their index). This call shortens the `ArrayList` by `(toIndex - fromIndex)` elements. (If `toIndex==fromIndex`, this operation has no effect.)
>
> This method is called by the `clear` operation on this list and its sublists. Overriding this method to take advantage of the internals of the list implementation can substantially improve the performance of the `clear` operation on this list and its subLists.
>
> This implementation gets a list iterator positioned before `fromIndex` and repeatedly calls `ListIterator.next` followed by `ListIterator.remove`, until the entire range has been removed. Note: If `ListIterator.remove` requires linear time, this implementation requires quadratic time.
>
> Parameters:
>
> | | |
> |---|---|
> | `fromIndex` | index of first element to be removed. |
> | `toIndex` | index after last element to be removed. |

這個函式對於 `List` 的終端用戶是沒有意義的。它的惟一目的是使 subclasses 能夠更容易地提供針對 sublists 而運作的快速函式`clear()`。如果沒有`removeRange()`，當 `clear()`被喚起並執行於 sublists 身上，subclasses 將承受「二次方時間」的糟糕效率，否則就得重寫全部的 `subList` 機制 — 那可不是件容易的事！

那麼，當你爲了日後繼承而設計一個 class 時，如何決定應該顯露（*expose*）哪些 protected 函式或欄位呢？這裡並沒有一條條可供遵循的神奇法則。你能夠做的最棒的事情就是努力思考，發揮最好的想像，然後撰寫一些 subclasses 測試你的想法。你應該儘量少提供 protected 函式和欄位，因爲它們每一個都代表對實作細節的某種承諾（譯註：也就形成一種約束）。但是另一方面，你又不能對它們供應太少，因爲遺漏某個 protected 函式也許就可能因此造成 class 的繼承沒有實際作用。

當你針對繼承設計出一個可能被廣泛使用的 class 時，你應該意識到你是對文件中說明的自用（self-use）形式承擔了一份永久義務，也是對隱含於 protected 函式和 protected 欄位內的實作細節承擔了一份永久義務。這些承諾使得後續版本很難（或甚至根本不可能）提高 class 的效率或功能。

另請注意，繼承所需的特殊文件可能會使標準文件變得混亂。標準文件是爲那些「創建 class 實體、呼叫實體相關函式」的程式員而設計的。撰寫本書此刻，很少有什麼工具或註解約定（commenting conventions）用來區分「一般的 API 文件」和「subclasses 程式員感興趣的資訊」。

class 還必須遵守其他一些限制規定，才能被用於繼承機制：無論直接或間接，建構式絕對不能呼叫可被覆寫函式（overridable methods）。如果違背這條原則，很可能導致程式失敗。這是因爲 superclass 建構式會在 subclass 建構式之前先執行起來，所以 subclass 中的覆寫函式會在 subclass 建構式執行前就被喚起。因此，如果那個覆寫函式倚賴 subclass 建構式（譯註：彼時尚未執行）所完成的任何初始化結果，該函式將無法如預期般地運作。爲了更具體說明，下面舉一個簡單例子，刻意違背上述原則：

```
public class Super {
    // Broken - constructor invokes overridable method
    public Super() {
        m();
    }

    public void m() {
    }
}
```

下面是個 subclass，覆寫了 `m()`。此函式將被 `Super` 的惟一建構式呼叫起來（那是不正確的）：

```
final class Sub extends Super {
   private final Date date; // Blank final, set by constructor

   Sub() {
      date = new Date();
   }

   // Overrides Super.m, invoked by the constructor Super()
   public void m() {
      System.out.println(date);
   }

   public static void main(String[] args) {
      Sub s = new Sub();
      s.m();
   }
}
```

你可能預期這個程式會列印日期兩次，但第一次它卻列印 null。這是因爲函式 `m()` 被建構式 `Super()`喚起，而彼時建構式 `Sub()`尚未有機會初始化 `date` 欄位。注意，這個程式觀察到一個 final 欄位（`date`）竟有兩個不同狀態（states）。

`Cloneable` 和 `Serializable` 兩介面給繼承設計帶來了特別的困難。對於專門訴諸繼承的 class 來說，實現上述任何一種介面都不是好主意，因爲這會給日後擴展它們的程式員添加一些負擔。你可以運用某些特殊手法，讓 subclasses 實現這些介面，而不需要扛起那些負擔。這些手法在**條款 10** 和**條款 54** 介紹。

如果你決定在一個爲繼承而設計的 class 中實現 `Cloneable` 介面和 `Serializable` 介面，由於 `clone()`和 `readObject()`的行爲很像建構式，因此受到同樣的約束：無論直接或間接，`clone()`和 `readObject()`都不能呼叫可被覆寫函式（overridable methods）。對 `readOnly()`而言，覆寫函式會在「subclass 的狀態被反序列化（*deserialized*）」之前先執行起來。對 `clone()`而言，覆寫函式會在「subclass 的 `clone()`修改克隆件的狀態」之前先執行起來。這兩種情況都可能導致程式失敗。就 `clone()`而言，這一失敗會破壞被克隆之物件（cloned object）和克隆件（clone）本身。

最後一點，如果你決定對專爲繼承而設計的 class 實現 `Serializable`，而其中有個 `readResolve()`或 `writeReplace()`，你必須令這兩個函式爲 protected 而不是 private。如果它們是 private，就會被 subclasses 悄然忽略。這正是「把實作細節變成 class API 的一部分，以允許發生繼承」的常見情況。

現在應該很清楚了，針對繼承而設計的 class，其身上會帶有某些實質限制。因此「允許繼承」並不是輕輕鬆鬆就可承諾的決定。某些情況下「允許繼承」的決定是對的，例如對於 abstract classes（包括 interface）的骨架性實作（skeletal implementation，**條款 16**）。其他情況下「允許繼承」的決定明顯錯誤，例如對於 *immutable* classes（**條款 13**）。

但對於一般的 concrete classes 情況又如何？傳統意義上它們既不是 final，也並非爲了 *subclassing* 而設計並帶相關文件。這種情況下繼承是危險的，因爲對這樣的 classes 所做的每一個改變都有可能使其客戶類別（client classes）崩潰。這不只是理論上的問題。現實之中，在修改一個 nonfinal concrete class（它並非爲繼承而設計）的內部後，收到「與 *subclassing* 相關的臭蟲提報」並不罕見。

解決這類問題的最好辦法是，對於那些並非「專門爲了被安全地子類別化（*subclassed*）並攜帶相關文件」的 class，禁止對它進行子類別化（*subclassing*）。禁止辦法有兩種，比較容易的作法是將 class 宣告爲 final。另一種作法是令所有建構式成爲 private 或 package-private，並添加一個 public static ***factories*** 取代建構式。後一種作法爲「內部運用 subclasses」提供了彈性，**條款 13** 曾有討論。兩種作法都可被接受。

上面的建議或許會引發爭論，因爲對一般的 concrete classes 進行 *subclassing* 動作，以便添加諸如 instrumentation（儀表功能，如計數器等）、notification（通知功能）、synchronization（同步功能）等等設施，甚或限制某些功能，這類作法對程式員來說已經愈來愈熟悉了。如果 class 實現了某些能夠體現其精髓的 interface，例如 `Set`、`List` 或 `Map`，那麼你不應該因爲禁止 *subclassing* 而感到惋惜。wrapper class（外覆類別，**條款 14**）爲我們帶來繼承之外的另一種「改變 class 功能」的良好選擇。

如果 concrete class 並未實現任何標準的 interface，那麼「禁止它被繼承」也許會給某些程式員帶來不便。如果你認爲你必須同意這樣的 class 被繼承，一個合理作法是保證這個 class 永遠不要呼叫其任何可被覆寫函式，並以文件清楚說明這個事實。換句話說，徹底消除 class 對於可被覆寫函式的自用性（self-use）。如果做到

了這一點，你就可以創建「能夠安全進行 *subclassing*」的 class 了。這時候，覆寫任何函式都不會影響其他函式的行爲。

你可以機械似地消除 class 對於其可被覆寫函式的自用性（self-use），而且不改變 class 的行爲。作法是把可被覆寫函式的「函式本體」搬移到一個 private 輔助函式中，並讓每一個可被覆寫函式呼叫其 private 輔助函式，然後直接喚起可被覆寫函式的 private 輔助函式，用以代替可被覆寫函式的每一個自用（self-use）動作。

條款 16：儘量以 interfaces（介面）取代 abstract classes（抽象類別）

Java 語言提供 interfaces 和 abstract classes 兩種機制，用以「定義一個型別並允許多種實作手法」。兩者之間最明顯的區別是：abstract classes 可以內含函式實作碼，interface 則完全不允許。更重要的區別是：如果要實現 abstract class 所定義的型別，必須先做出該 abstract class 的一個 subclass。但是任何 class 只要定義所有必要函式，並遵守一般契約，都得以實現一個 interface，不論該 class 位於 class 繼承體系何處。由於 Java 只允許單一繼承（single inheritance），因此 abstract classes 身上的約束嚴重限制了它們「作爲一個型別定義」的用途。

你可以輕易翻新一個現有 classes，令它實現一個新的 interface。惟一要做的工作就是添加現有 class 之中並不存在的函式，並在 class 宣告式中增加一個 `implement` 子句。例如，當 `Comparable` 介面被引入 Java 平台，許多既有的 classes 於是被翻新以實現這個介面。然而通常既有的 classes 無法翻新擴展一個新的 abstract class。如果你要讓兩個 classes 擴展同一個 abstract class，必須把這個 abstract class 放在型別體系（type hierarchy）內該兩個 classes 的上端，成爲其共同祖類別（ancestor）的一個子類別（subclass）。不幸的是，這會對型別體系造成巨大破壞，迫使該共同祖先的所有後代都得擴展新的 abstract class — 不論合適與否。

Interface 是定義 mixins（混合型別）的理想選擇。所謂 **mixin** 是：class 除了主要型別之外還可實作的另一個（次要）型別，用以宣告 class 提供了某些可選行爲（譯註：由於 Java 不支援多重實作繼承，因此這原本是辦不到的）。例如 `Comparable` 就是個 mixin 介面，允許 class 宣告「其實體(s)將根據其他可比較物件（comparable objects）進行排序」。這樣的介面我們稱之爲 mixin，因爲它允許型別的「可選機能」和「主要機能」混合在一起。Abstract classes 不能被用來定義 mixin，理由同前：它們不能被翻新至現有的 classes 之上，因爲 class 不能有一個以上的 parent classes，而 class 繼承體系中也沒有合理放置 mixin 的地方。

Interfaces 使我們得以建構出 nonhierarchical type frameworks（非階層體系的型別框架）。某些事物可以利用 type hierarchies 組織得很好，但另一些事物恐怕無法清晰組織成嚴密的 type hierarchies。舉個例子，假設我們有一個 interface 代表歌手，另一個 interface 代表詞曲作者：

```
public interface Singer {
```

```
    AudioClip sing(Song s);
}
public interface Songwriter {
    Song compose(boolean hit);
}
```

現實生活中，有些歌手身兼作詞作曲者。由於我們使用 interfaces 而不是使用 abstract classes 來定義這些型別，所以 class 同時實現 `Singer` 和 `Songwriter` 是可以的。事實上我們可定義第三個 interface，令它擴展 `Singer` 和 `Songwriter` 並添加適當的新函式：

```
public interface SingerSongwriter extends Singer, Songwriter {
    AudioClip strum();
    void actSensitive();
}
```

你並不一定總是需要這樣的彈性，但一旦你需要，interfaces 就成了你的救生圈。另一種作法是完成一個膨脹的 class 繼承體系，其中針對每一個「受到支援的屬性組合（combination of attributes）」對應一個 class。如果型別系統中有 n 個屬性，你也許需要支援 2^n 個可能組合。這是所謂的組合爆炸（combinatorial explosion）。膨脹的 class 繼承體系會導致膨脹的 classes — 這種 class 內含大量「彼此之間只是引數型別不同」的函式，這是因爲在 class 繼承體系中並沒有哪個型別可用來表示共同的行爲。

藉由 wrapper class（外覆類別，**條款 14**）手法，interfaces 使得程式機能得以安全而強大地獲得提昇。如果你使用 abstract classes 來定義型別，那麼當程式員需要添加功能時，除了繼承之外別無選擇，而由此獲得的 classes 和 wrapper class 相比，功能比較少、也比較脆弱。

儘管 interfaces 不允許含有函式實作碼，但使用 interfaces 來定義型別並不妨礙你爲程式員提供實作上的幫助。你可以結合 interfaces 和 abstract classes 的長處：爲每一個你所匯出（*export*）的有所作爲的（*nontrivial*）interface 提供一個「抽象骨幹實作類別」（abstract skeletal implementation class）。interface 還是用來定義型別，「骨幹實作類別」則承擔所有實作任務。

習慣上我們把「骨幹實作類別」命名爲 `Abstract`*`Interface`*，其中的 *`Interface`* 是實現之介面名稱。例如 Collections Framework 就爲每一個主要的 collection interface 提供了一份「骨幹實作類別」：`AbstractCollection`、`AbstractSet`、`AbstractList` 和 `AbstractMap`。

如果設計正確，「骨幹實作類別」會使得程式員極容易爲 interface 提供實作碼。例如下面是一個 static ***factory*** 函式，內含一份完整、全功能的 `List` 實作品：

```
// List adapter for int array
static List intArrayAsList(final int[] a) {
    if (a == null)
        throw new NullPointerException();

    return new AbstractList() {
        public Object get(int i) {
            return new Integer(a[i]);
        }

        public int size() {
            return a.length;
        }

        public Object set(int i, Object o) {
            int oldVal = a[i];
            a[i] = ((Integer)o).intValue();
            return new Integer(oldVal);
        }
    };
}
```

當你思考一個 `List` 實作品應該爲你做些什麼的時候，本例展示的「骨幹實作類別」的強大威力應該會令你印象深刻。順帶一提，這個例子可視爲一個 ***Adapter*** [Gamma95, p139]，允許一個 `int` array 被視爲一個 `Integer` list。但由於 `int` 和 `Interger` 之間的來回轉換，效率並不很好。注意，本例提供了一個 static ***factory***，而且本例的 class 是一個隱藏於 static ***factory*** 之內、不可存取的匿名類別（inaccessible anonymous class, **條款 18**）。

「骨幹實作類別」的漂亮在於它們提供了 abstract classes 的實作支援，卻不必接受 abstract classes 被當做型別定義時需要接受的嚴厲束縛。對大多數 interface 實作者而言，擴展「骨幹實作類別」是顯而易見的抉擇，但也僅僅是個選擇而已。如果預先存在的 class 無法擴展「骨幹實作類別」，我們總還可以手工實現 interface。此外，「骨幹實作類別」還可以協助實作者完成任務。負責實現 interface（假設爲 `I`）的那個 class（假設爲 `C`），可以將「對 `I` 函式的呼叫動作」轉發（*forward*）給 `C` 所擁有的一個負責擴展「骨幹實作類別」的 private inner class 的實體。這項技術

我們稱爲「模擬多重繼承」（simulated multiple inheritance），與**條款 14** 所討論的外覆類別手法（wrapper class idiom）十分類似。它提供多重繼承的大部分優點，並避免其缺陷。

撰寫「骨幹實作類別」是十分簡單的事，近乎冗長乏味。首先你必須分析 interface，確定哪些函式是其他函式賴以生存的基本元素。這些基本元素應該成爲你的「骨幹實作類別」中的 abstract methods（抽象函式）。然後你必須爲 interface 內的所有其他函式提供具象實作（concrete implementations）。下面是 `Map.Entry` interface 的一個「骨幹實作類別」。本書撰寫之際，這個 class 尚未被加入 Java 標準程式庫，但它或許應該被加入：

```
// Skeletal Implementation
public abstract class AbstractMapEntry implements Map.Entry {
   // Primitives
   public abstract Object getKey();
   public abstract Object getValue();

   // Entries in modifiable maps must override this method
   public Object setValue(Object value) {
      throw new UnsupportedOperationException();
   }

   // Implements the general contract of Map.Entry.equals
   public boolean equals(Object o) {
      if (o == this)
         return true;
      if (! (o instanceof Map.Entry))
         return false;
      Map.Entry arg = (Map.Entry)o;

      return eq(getKey(),   arg.getKey()) &&
            eq(getValue(), arg.getValue());
   }

   private static boolean eq(Object o1, Object o2) {
      return (o1 == null ? o2 == null : o1.equals(o2));
   }

   // Implements the general contract of Map.Entry.hashCode
   public int hashCode() {
      return
         (getKey()   == null ? 0 :   getKey().hashCode()) ^
         (getValue() == null ? 0 : getValue().hashCode());
   }
}
```

由於「骨幹實作類別」被設計用於繼承，因此你應該遵守**條款 15** 關於設計和文件方面的所有規則。爲求簡潔，先前例子中我省略了該有的文件註解，但是良好的文件絕對是「骨幹實作類別」的必備物。

運用 abstract classes 來定義「允許多份實作」的型別，比起運用 interface，有一個明顯的優勢：演進（發展）abstract class 比演進（發展）interface 更容易。如果日後你打算爲 abstract class 添加一個新函式，你總是可以添加一個內含合理預設實作碼的具象函式（concrete method）。於是該 abstract class 的每一份現有實作品都會供應這個新函式。如果你用的是 interface，得不到這樣的結果。

一般來說，爲 public interface 添加函式，而又不破壞該 interface 的所有客戶，幾乎是不可能的。先前實現該 interface 的那些 classes 將會因爲缺少這個新函式而無法通過編譯。你可以在「爲 interface 添加新函式」的同時也爲「骨幹實作類別」添加相同的新函式，藉此多少減輕可能的破壞。但這其實不能解決問題。任何實作品如果並非繼承自「骨幹實作類別」，還是會遭到破壞。

所以，你必須認真設計 public interfaces。一旦 interface 被發佈並被廣泛使用之後，想要再改變它是不可能的。你必須第一次就正確設計好 interface。即使 interface 內含極小的缺陷，也會使你（設計者）和使用者永遠蒙受影響。如果 interface 極不完善，可能導致 API 徹底失敗。發佈一個嶄新 interface 時，你能夠做的最好事情就是在它安定下來之前，讓儘可能多的程式員以儘可能多的方式來實作它。這使你有機會在還可以更正 interface 的時候發現它的缺陷。

總而言之，interface 通常是定義「允許多份實作同時存在」的型別的最佳辦法。但是當我們認爲「演進的容易性」比「程式的彈性和功能」更重要時，上一句話就不成立。這時候你應該使用一個 abstract class 來定義型別 — 前提是你理解並接受它所帶來的束縛。如果你匯出（*export*）一個有所作爲的（*nontrivial*）interface，你應該強烈考慮爲它提供一個「骨幹實作類別」。最後一點，你應該極爲謹慎地設計你的每一個 public interfaces，並透過多份實作碼對它們做全面測試。

條款 17：interfaces 只應當被用來定義型別（types）

當 class 實現某個 interface，該 interface 就成了一個型別，可被用來指涉（指向）這個 class 的實體。一個「實現某 interface」的 class 應該說清楚客戶可對其實體做些什麼動作。爲任何其他目的而定義 interface，都是不恰當的。

有一種 interface 並不滿足上述說法，那是所謂的 **constant interface**（常數介面）。這種 interface 不含函式，僅由 static final 欄位組成，每一個欄位匯出（*exporting*）一個常數。打算使用這些常數的 class 如果實現此一 interface，便可不必在常數名稱之前冠以 class 名稱。下面是個例子：

```
// Constant interface pattern - do not use!
public interface PhysicalConstants {
    // Avogadro's number (1/mol)
    static final double AVOGADROS_NUMBER   = 6.02214199e23;

    // Boltzmann constant (J/K)
    static final double BOLTZMANN_CONSTANT = 1.3806503e-23;

    // Mass of the electron (kg)
    static final double ELECTRON_MASS      = 9.10938188e-31;
}
```

constant interface 其實是對 interface 的一種不良運用。「class 內部使用某些常數」只是一種實作細節。「實現 constant interface」會導致實作細節洩漏至 class 的 exported API 中。「實現 constant interface」對 class 用戶並不重要，反而可能混淆其用戶。更糟的是它代表了一種承諾：如果日後版本的 class 被改爲不再使用常數，爲了保持二進制相容，class 仍需實現該 interface。如果一個 final class 實現了一個 constant interface，其所有 subclasses 會因爲 interface 中的常數而造成命名空間污染（namespaces pollution）。

Java 標準程式庫中有一些 constant interfaces，例如 `java.io.ObjectStreamConstants`。你應該視這些 interfaces 爲反常，不要模仿它們。

如果需要匯出（*export*）常數，另有一些合理選擇。如果常數與既有的 class 或 interface 緊密關聯，那就應該把它們加入那個 class 或 interface 中。例如 Java 標準程式庫的所有 numerical wrapper classes（數值外覆類別）如 `Integer` 和 `Float` 等等都匯出 `MIN_VALUE` 和 `MAX_VALUE` 兩個常數。如果常數適合作爲列舉型（enumerated type）成員，你應該以一個 typesafe enum class（**條款 21**）匯出它們，否則應該以一個「不

可被實體化」的 utility class（**條款 3**）加以匯出。下面的例子是先前那個 `PhysicalConstants` 的 utility class 版本：

```
// Constant utility class
public class PhysicalConstants {
  private PhysicalConstants() { }  // Prevents instantiation

  public static final double AVOGADROS_NUMBER   = 6.02214199e23;
  public static final double BOLTZMANN_CONSTANT = 1.3806503e-23;
  public static final double ELECTRON_MASS      = 9.10938188e-31;
}
```

儘管這個 utility class 版本的 `PhysicalConstants` 要求客戶必須以 class 名稱來修飾常數名稱，但這對 API 而言只是很小的代價。雖然 Java 語言最終有可能允許匯入（*import*）static 欄位，但此同時，你可以將經常使用的常數保存在區域變數或 private static 欄位中，藉以降低過度的文字鍵入（譯註：意指上面所說每個常數都必須以 class 名稱加以資格修飾），例如：

```
private static final double PI = Math.PI;
```

總而言之，interfaces 應該只被用來定義型別（types）。它們不應被用來匯出（*exports*）常數。

條款 18：優先考慮 static member classes，然後才是 nonstatic

nested class（巢狀類別）是一種定義於其他 class 內部的 class；它應該僅僅爲其 enclosing class（外圍類別）而存在。如果 nested class 在其他場合也有用，你應該把它變成一個 top-level class(上層類別)。Nested classes 共有四種：(1) **static member classes**（靜態成員類別）、(2) **nonstatic member classes**（非靜態成員類別）、(3) **anonymous classes**（匿名類別）、(4) **local classes**（區域類別）。後三種又被稱爲 **inner classes**（內隱類別）。本條款將告訴你它們的使用時機和原因。

static member class 是其中最簡單的一種。它很適合被視爲一般 class，只不過宣告於其他 class 內部，因而可以存取其外圍 class 的所有成員，包括 private 成員。static member class 是其外圍 class 的一個 static 成員，遵循其他 static 成員一樣的可存取性（accessibility）；如果它被宣告爲 private，就只能在外圍 class 之內被存取。

static member class 的常見用法是作爲一個輔助用的 public class，只在與其外圍 class 結合時才有用。例如一個「用以描述計算器所支援之操作種類」的 typesafe enum（**條款 21**）。我們應該設計 `Operation` 使它成爲 `Calculator` 的一個 public static member class。`Calculator` 的客戶可以使用諸如 `Calculator.Operation.PLUS`、`Calculator.Operation.MINUS` 等形式來引用這些操作。稍後另有介紹。

語句構造上，static 和 nonstatic member classes 的惟一區別是，前者在其宣告式中用到飾詞 static。然而儘管語句形式相近，這兩種 nested classes 區別很大。每一個 **nonstatic member class** 和其外圍 class 的實體之間有種隱隱關聯。在 nonstatic member class 的 *instance* 函式中喚起外圍實體的函式，或運用資格修飾後的（*qualified*）`this` 構件取得一個 reference 指向外圍實體 [JLS,15,8.4]，都是可能的。如果 nested class 的實體可在其外圍 class 實體之外單獨存在，這個 nested class 就不該成爲一個 nonstatic member class，因爲在缺乏外圍實體的情況下創建一個 nonstatic member class 實體是不可能的。

nonstatic member class 的實體和其外圍實體之間的關聯在前者創建之初就建立好了，此後無法修改。正常情況下這種關聯會因爲「外圍 class 的一個 *instance* 函式呼叫 nonstatic member class 建構式」而被自動建立起來。如果你想使用 `enclosingInstance.NewMemberClass(args)`手動建立這個關聯也可以，但很少

如此。一如你所預期，這個關聯會在 nonstatic member class 實體中佔用空間，並增加其建構過程所需時間。

nonstatic member class 常被用來定義 ***Adapter*** [Gamma95, p139]，允許我們將外圍 class 的實體視爲某些無親屬關係的 class 的實體。例如 `Map` 實作品通常使用 nonstatic member classes 實現其 collection views（群集映件），該物件由 `Map` 的 `keySet()`、`entrySet()`、`values()`等函式傳回。同樣道理，其他 collection interfaces（群集介面）如 `Set` 和 `List` 通常也使用 nonstatic member classes 實現其迭代器（iterators）：

```
// Typical use of a nonstatic member class
public class MySet extends AbstractSet {
    ... // Bulk of the class omitted

    public Iterator iterator() {
        return new MyIterator();
    }

    private class MyIterator implements Iterator {
        ...
    }
}
```

如果你宣告一個 member class 而它並不要求存取外圍實體，切記在其宣告式中使用 static 飾詞，使它成爲一個 static member class 而不是一個 nonstatic member class。如果忽略 static 飾詞，那麼其每一個實體將內含一個 reference 指向外圍物件。維護這個 reference 需要時間和空間成本，而又得不到好處。你曾經配置「無外圍實體」的一個 nonstatic member class 實體嗎？不可能，因爲 nonstatic member class 的實體必須要有一個外圍實體。

private static member classes 一般用來表示其外圍物件的組件。例如，考慮一個將 keys 和 values 關聯起來的 `Map` 實體。`Map` 實體通常針對每一個 key-value pair 準備一個內部的 `Entry` 物件。儘管每個 entry（條目）與 map 互有關聯，但作用於 entry 之上的函式 `getKey()`、`getValue()`、`setValue()`卻不需要存取 map。因此以 nonstatic member class 表現 entry 未免浪費；以 private static member class 表現它最合適。如果你在宣告 entry 時遺漏 static 飾詞，map 還是能夠有效運作，但每個 entry 之內將包含一個指向外圍 map 的 reference，那是多餘的，形成空間和時間的浪費。

如果我們所討論的 class 是某個 exported class 的一個 public 或 protected 成員，那麼正確選擇 static 或 nonstatic member class 便加倍重要了。因爲這種情況下的 member class 是 exported API 的元素之一，日後，除非破壞二進制相容性，否則無法將 nonstatic member class 改變爲 static member class。

Anonymous classes（匿名類別）與 Java 語言中的任何構件都不相像。一如你所預期，這種 class 沒有名稱。它並不是某個外圍 class 的成員；它並不與其他成員一起宣告，而是在被使用時刻才同時被宣告、被實體化。只要算式合法，anonymous classes 可出現於程式任何地點。其行爲相仿於 static 或 nonstatic member classes：如果出現於 nonstatic 環境（context）則擁有外圍實體，否則就不擁有外圍實體。

Anonymous classes（匿名類別）的應用有些限制。由於它同一時間被宣告、被實體化，因此它只能被使用於程式碼將它實體化的那一個點上。Anonymous classes 沒有名稱，所以實體化之後不能再被引用。這種 classes 通常只實作其 interface 或 superclass 中的函式，並不宣告任何新函式，因爲程式中沒有任何「可指名的型別」（nameable type）可存取那些新函式。這種 classes 出現在算式（expressions）之間，因此應該簡短些，可能 20 行左右或更少。過長的 anonymous classes 會傷害程式碼的可讀性。

Anonymous classes 的常見用途之一是創建「函式物件」（function object），例如創建一個 `Comparator` 實體。以下函式根據 string 的長度對一個 strings array 排序：

```
// Typical use of an anonymous class
Arrays.sort(args, new Comparator() {
    public int compare(Object o1, Object o2) {
        return ((String)o1).length() - ((String)o2).length();
    }
});
/* 譯註：本例灰色部分便是 anonymous class 的定義，意思是產生一個實現 Comparator
interface 的 class object。Comparator 源碼如下（參見 java.util.Comparator）：
public interface Comparator {
    int compare(Object o1, Object o2);
    boolean equals(Object obj);
}
*/
```

Anonymous classes 的另一個常見用途是創建「行程物件」（process object），例如 `Thread`、`Runnable` 或 `TimerTask` 實體。第三個常見用途是給 static ***factory*** 函式使用（參見**條款 16** 的 `intArrayAsList()`）。第四個常見用途是在精巧複雜的 typesafe enums 的 public static final 欄位初始式（initializers）中工作（見**條款 21** 的 `Operation`）。如果 `Operation` class 是 `Calculator` 的一個 static member class（一

如先前所建議），那麼各個 Operation 常數都是雙層巢狀（doubly nested）classes：

```
// Typical use of a public static member class
public class Calculator {
   public static abstract class Operation {
      private final String name;

      Operation(String name)   { this.name = name; }

      public String toString() { return this.name; }

      // Perform arithmetic op represented by this constant
      abstract double eval(double x, double y);

      // Doubly nested anonymous classes
      public static final Operation PLUS = new Operation("+") {
         double eval(double x, double y) { return x + y; }
      };
      public static final Operation MINUS = new Operation("-") {
         double eval(double x, double y) { return x - y; }
      };
      public static final Operation TIMES = new Operation("*") {
         double eval(double x, double y) { return x * y; }
      };
      public static final Operation DIVIDE = new Operation("/") {
         double eval(double x, double y) { return x / y; }
      };
   }

   // Return the results of the specified calculation
   public double calculate(double x, Operation op, double y) {
      return op.eval(x, y);
   }
}
```

所有四種 nested classes 中，local class 或許是最少被使用的。它可以被宣告於 local（區域）變數能夠被宣告的任何地點，並且和 local 變數遵守相同的作用域規則（scoping rule）。Local classes 與其他三種 nested classes 在某些屬性上一致：它們和 member classes 一樣也有名字，可被重複使用。它們和 anonymous classes 一樣，若且惟若（*only and only if*）用於 nonstatic 環境中，則擁有外圍實體。它們也和 anonymous classes 一樣，應該簡短，才不至於傷害外圍函式或初始式（initializer）的可讀性。

讓我再扼要說一遍。共有四種不同的 nested classes，各有用處。如果 nested class 需要在某函式之外可被看見，或如果它太長而不適合塞入一個函式內，請使用 member class。如果 member class 實體需要一個 reference 指向其外圍實體，請讓它成爲 nonstatic；否則請讓它成爲 static。假設 class 存在於一個函式內，如果你只需在惟一一個位置上創建實體，並且有一個既存的型別可以描繪出該 class 的特徵，那麼請把它做成一個 anonymous class，否則請做成一個 local class。

10

序列化/次第讀寫

Serialization

譯註：我曾經在《*Thinking in Java*, 2e 中文版》中將 "serialization" 根據技術意義譯爲「次第讀寫」。坊間另一可見譯詞是「序列化」。鑒於本書大量出現 "serialized" 和 "deserialized"，譯爲「序列化」和「反序列化」較易表達，故改採之。中文用詞無關緊要，因本書與《*Thinking in Java*, 2e 中文版》對術語 "serialization" 絕大多數採用英文原詞。謹此。

本章關心的主題是物件的 serialization APIs，它們提供了一個框架（framework），讓你將物件編碼（*encoding*）爲 byte streams，並得被再建構（*reconstructing*）爲物件。將物件編碼爲 byte stream 的動作是所謂 "serializing"（序列化）；反向過程則是所謂 "deserializing"（反序列化）。一旦物件被 serialized，其編碼結果可以從某個虛擬機器傳輸至另一個虛擬機器上，或儲存於磁碟中以便稍後被 deserialization。Serialization 提供了遠程通訊之「標準連線層物件表述」（standard wire-level object representation），以及 JavaBeans™ 組件架構（component architecture）之「標準永續資料格式」（standard persistent data format）。

條款 54：審慎實現 `Serializable`

想要讓一個 class 實體（物件）被 serialized，只需爲其宣告式加上 "`implements Serializable`" 即可。這個動作太簡單，導致某種誤解，以爲程式員幾乎不需要爲 serialization 做什麼努力。真象遠爲複雜得多。雖然「讓一個 class 成爲 serializable」的立即成本小到可以忽略，但長期成本往往很大。

「實現 `Serializable`」所必須付出的主要成本（代價）是，它會使「改變 class 實作」的彈性降低 — 在 class 發行（*released*）之後。一旦某個 class 實現了 `Serializable`，其 byte-stream 編碼結果（或謂 serialized form）就會變成 class exported API 的一部分。一旦你將這個 class 廣泛傳佈出去，往往就必須長期（甚至永久）支援該 serialized form，就像你必須支援 exported API 的所有其他部分一樣。如果

你沒有設計自己的 serialized form，只是接受預設格式，這個格式將永遠和該 class 的原始內部表述（original internal representation）相繫。換句話說如果你接受了預設的 serialized form，這個 class 的 (1) `private` 和 (2) package-`private` 的「*instance* 欄位」會變成其 exported API 的一部分，而「將欄位可存取性最小化」（**條款 12**）的實踐準則也將因此喪失「資訊隱藏」的效用。

如果你接受了預設的 serialized form，並於日後改變 class 的內部表述（internal representation），可能就會造成 serialized form 的不相容。客戶如果使用舊版的 class 完成 serialize 動作，卻企圖以新版 class 完成 deserialize 動作，會造成程式失敗。是有可能在維護原本 serialized form 的情況下改變 class 的內部表述（使用 `ObjectOutputStream.putFields()`和 `ObjectInputStream.readFields()`），但技術上比較困難而且會在源碼留下一些明顯的「瘤」。因此你應該謹慎設計一個你願意長期使用的高品質 serialized form（**條款 55**）。這麼做雖然會增加開發成本，但值得！設計精良之 serialized form 可能會對 class 的演化帶來一些束縛；設計不良之 serialized form 則根本使 class 僵固無法演化。

這裡有一個例子可以說明 serializability 伴隨而來的演化束縛。"*stream unique identifiers*"，通常亦被稱爲 serial version UIDs，是每個 serializable class 都會擁有的獨一無二識別號。如果你未曾「宣告一個名爲 `serialVersionUID` 的 private static final `long` 欄位」來明確指出這個識別號，系統會以一個繁複程序自動爲你的 class 生成一個，其值受到 class 名稱、class 所實現之 interfaces 名稱、class 的所有 public 和 protected 成員名稱的影響。如果你改變上述任何一個名稱，例如爲求便捷而添加一個（即使平淡無奇的）函式，前述自動生成之 serial version UID 也會改變。因此如果你未明確宣告一個 serial version UID，相容性便被打破。

實作 `Serializable` 的第二個成本是，它會增加臭蟲和安全漏洞的可能性。正常情況下物件以建構式（constructors）創建出來，serialization 則是一個超脫語言之外的「物件創建」機制。無論你接受 deserialization 的預設行爲或覆寫它，它都是一個「隱藏式建構式」，它身上該注意的問題和一般建構式完全相同。由於它並非明顯的建構式，你很容易忘記你必須確保 deserialization 能夠保證「被真正建構式建立的所有約束條件（invariants）」，你也很容易忘記 deserialization 不得允許「攻擊者」存取尚在建構過程中的物件的內部資料。倚賴預設的 deserialization 機制，很容易使物件的約束條件受到破壞，以及蒙受非法存取（**條款 56**）。

實作 `Serializable` 的第三個成本是，它會增加「發佈新版 class」時的測試負擔。當一個 serializable class 被改變，很重要的一點是檢查是否能夠以新版本 *serialize* 一個實體，並以舊版本完成 *deserialize* 動作。反之亦然。測試量與「serializable classes

的數量和產品發行量的乘積」成正比，那可能是個巨大數字。這些測試無法自動化進行，因爲除了二進制相容外，你還必須測試語意（semantic）相容。換句話說你必須同時確保 serialization-deserialization 成功並導致原物忠實重現。一個 serializable class 的改變愈大，你就需要對它進行愈多測試。如果你在初始設計一個 class 時就能夠謹慎地爲它設計一個自定的 serialized form（**條款 55**），上述測試需求就可以大大降低（但無論如何不可能完全消失）。

「實現 `Serializabl`」並不是一個可輕率做出的決定。實現後可以提供一些實際利益：如果 class 參與某些「倚賴 serialization 進行物件傳輸或永續化」的 framework，「實現 `Serializabl`」就是必要的。此外如果 class A 被用做 class B 的組件，而 B 必須實現 `Serializable`，那麼 A「也實現 `Serializabl`」將有助於兩者的合作。然而「實現 `Serializable`」會帶來許多成本，請謹慎權衡其中利弊。根據經驗，數值型 classes 如 `Date` 和 `BigInteger` 應該實現 `Serializable`，大多數 collection classes 也是。用以表現「活動物體」（active entities）之 classes 如 thread pools 則很少實現 `Serializable`。Java 1.4 提供了一個 XML-based JavaBeans 永續機制，所以你的 Beans 不再需要實現 `Serializable`。

「爲繼承而設計之 classes」（**條款 15**）很少需要實現 `Serializable`，interfaces 也很少需要擴展 `Serializable`。如果違反這個規則，會造成擴展或實現該 interface 的那些程式員的沉重負擔。然而有時候違反這個規則是適當的，例如當一個 class 或 interface 的存在主要是爲了參與某些 framework，而後者要求所有參與者都必須實現 `Serializable`，那麼爲這些 class 或 interface 實現或擴展 `Serializable` 當然就是完全合理的。

這裡有一條警告與「無法實現 `Serializable`」有關。如果「爲繼承而設計之 class」並非 serializable，那麼很可能無法爲它寫一個 serializable subclass。更具體地說，上述所謂「無法」是指如果 superclass 不提供一個可取用之「無參數建構式」的話。因此你應該考慮針對「爲繼承而設計之 nonserializable class」提供一個無參數建構式。通常這並不需要額外心力，因爲許多「爲繼承而設計之 class」是無狀態的（no state。譯註：亦即無資料欄位），但這並非絕對。

如果能夠在所有約束條件（invariants）已經建立的情況下再來創建物件，那是最理想的了（**條款 13**）。如果建立這些約束條件時需要客戶端提供資訊，實際上等於排除了無參數建構式的使用。天真爛漫地爲 class 增加一個無參數建構式和一個初始化函式，卻仍由 class 的其他建構式來建立約束條件，會使 class 的狀態空間（state-space）複雜化，增加出錯可能。

以下我將示範爲 nonserializable extendable class 添加一個無參數建構式，並避免上述缺陷。假設 class 有建構式如下：

```
public AbstractFoo(int x, int y) { ... }
```

下面的轉化版本添加一個「protected 無參數建構式」和一個「初始化函式」，後者有著和正常建構式相同的參數，用以建立相同的約束條件：

```
// Nonserializable stateful class allowing serializable subclass
public abstract class AbstractFoo {
    private int x, y; // The state
    private boolean initialized = false;

    public AbstractFoo(int x, int y) { initialize(x, y); }

    /**
     * 這個建構式和後面的函式允許 subclass 的
     * readObject() 初始化我們的內部狀態（internal state）。
     */
    protected AbstractFoo() { }

    protected final void initialize(int x, int y) {
        if (initialized)
            throw new IllegalStateException(
                "Already initialized");
        this.x = x;
        this.y = y;
        ... // 這裡做原建構式（original constructor）所做的任何事情
        initialized = true;
    }

    /**
     * 這些函式提供對內部狀態（internal state）的存取途徑，所以這個 class
     * 可以被 subclass 的 writeObject()手動（而非自動）完成 serialized。
     */
    protected final int getX() { return x; }
    protected final int getY() { return y; }

    // 以下必須被所有 public instance 函式呼叫。
    private void checkInit() throws IllegalStateException {
        if (!initialized)
            throw new IllegalStateException("Uninitialized");
    }
    ... // Remainder omitted
}
```

`AbstractFoo`的所有 *instance* 函式必須在進行自己的工作前先呼叫`checkInit()`。這可以確保萬一某個撰寫不當的 subclass 在「初始化某一實體」失敗時，上述的初始化函式會很快速而「乾淨地」失敗（譯註：而不至於拖泥帶水）。有了這樣的機制，我們就可以直率實作出一個 serializable subclass：

```
// Serializable subclass of nonserializable stateful class
public class Foo extends AbstractFoo implements Serializable {
   private void readObject(ObjectInputStream s)
         throws IOException, ClassNotFoundException {
      s.defaultReadObject();

      // Manually deserialize and initialize superclass state
      int x = s.readInt();
      int y = s.readInt();
      initialize(x, y);
   }
   private void writeObject(ObjectOutputStream s)
         throws IOException {
      s.defaultWriteObject();

      // Manually serialize superclass state
      s.writeInt(getX());
      s.writeInt(getY());
   }
   // Constructor does not use any of the fancy mechanism
   public Foo(int x, int y) { super(x, y); }
}
```

Inner classes（**條款 18**）應該很少實現 `Serializable`。它們使用由編譯器生成的合成欄位來儲存「用以指向外圍實體（enclosing instances）」的 references，以及儲存來自外部作用域（enclosing scopes）的區域變數值。這些欄位如何對應至 class 定義式，目前未有具體說明，就像 anonymous classes（匿名類別）和 local classes 的名稱一樣。因此我們並不清楚 inner class 預設的 serialized form 的定義。然而一個 static member class 可以實現 `Serializable`。

總而言之，「實現 `Serializable`」看似輕易其實不然。除非一個 class 在被使用一小段時間後就要被丟棄，否則「實現 `Serializable`」是一個嚴肅的承諾，應該謹慎待之。面對「爲繼承而設計」的 class 更是需要額外小心。面對這樣的 classes，其 subclasses 究竟可實現 `Serializable` 或被禁止 `Serializable`，兩者的分水嶺在於該 class 是否提供了一個「可取用的無參數建構式」（accessible parameterless constructor）。如果有，就允許（但不強制）subclasses 也實現 `Serializable`。

條款 55：考慮使用自定的序列化格式（serialized form）

如果你在時間緊迫下設計 class，一般合理做法是把心力集中於設計最好的 API。有時這意味發佈一個用後即丟的實作品，你知道很快你便會以一個新版本取代它。通常這不會形成問題，但如果這個 class 實現了 `Serializable` 並使用預設的 serialized form，你將無法完全擺脫那個用後即丟的倉促作品。它將永遠受到 serialized form 的牽制。這並不是純理論問題，它的確曾經發生在 Java 程式庫的數個 classes（例如 `BigInteger`）身上。

千萬不要在尚未考慮合適與否的情況下便貿然接受預設的 serialized form。你應該在神志清醒的狀態下判斷，並認爲如此所得的編碼結果從彈性、效率、正確性等觀點來看都適當而合理，才加以接受。一般而言，只有當你所選擇的（自定）編碼型式與預設的 serialized form 有極大相同度時，才應該接受預設的 serialized form。

物件的預設 serialized form，是「以該物件爲根之物件圖（object graph）」的實際表述（physical representation）的一個適度合理的高效編碼。換句話說，它描述出物件內含的資料，以及該物件所能觸及（reachable）的每一個其他物件。它也描述所有這些彼此連結（interlinked）的物件所形成的拓樸關係（topology）。理想的 serialized form 只含物件所表述的「邏輯資料」，有別於物件的實際表述（physical representation）。

如果物件的實際表述（physical representation）和其邏輯內容（logical content）完全相同，那麼預設的 serialized form 對它而言是適當的。例如預設之 serialized form 對以下 class 而言合理，此 class 用來表述「人名」：

```
// Good candidate for default serialized form
public class Name implements Serializable {
    /**
     * Last name.  Must be non-null.
     * @serial
     */
    private String lastName;

    /**
     * First name.  Must be non-null.
     * @serial
     */
    private String firstName;
```

```
    /**
     * Middle initial, or '\u0000' if name lacks middle initial.
     * @serial
     */
    private char   middleInitial;

    ... // Remainder omitted
}
```

從邏輯角度看，「姓名」由代表「名」（first name）和「姓」（last name）的兩個字串、以及表示「中間名大寫首字母」的一個字元組成。`Name` 之中的 *instance* 欄位清晰而明確地反映出這些邏輯內容。

縱使你認爲預設之 serialized form 是合適的，往往還必須提供一個 `readObject()` 以確保約束條件（invariants）和安全性（security）。以 `Name` 爲例，`readObject()` 可以確保 `lastName` 和 `firstName` 都是 non-`null`。這個主題由**條款 56** 詳細討論。

注意，雖然 `lastName`、`firstName`、`middleInitial` 欄位都是 private，它們還是有對應的文件註解。那是因爲這些 private 欄位定義了一個 public API，亦即這個 class 的 serialized form，而這個 public API 必須有說明文件。`@serial` 標籤用來告訴 Javadoc 工具程式把這些說明放進一個用來闡述 serialized forms 的特殊頁面中。

現在，考慮與 `Name` 大不相同的另一個例子，其中表現一個以字串形成的 list（讓我們暫時忘記「最好使用標準程式庫之 `List` 實作品」這條建議）：

```
// Awful candidate for default serialized form
public class StringList implements Serializable {
    private int size = 0;
    private Entry head = null;

    private static class Entry implements Serializable {
        String data;
        Entry  next;
        Entry  previous;
    }

    ... // Remainder omitted
}
```

從邏輯上說，這個 class 表現出一系列字串。實際上它表現的是個 doubly linked list。如果你接受預設的 serialized form，那將煞費苦心地鏡射（*mirror*）出 linked list 的每一筆記錄（entry）以及各記錄之間的所有雙向連結（links）。

當物件的實際表述（physical representation）和邏輯資料有實質差別時，如果你對此物件使用預設之 serialized form，會產生以下四個缺點：

- 它會永久性地將 exported API 繫縛（*ties*）於內部表述上。就本例而言，private `StringList.Entry` class 會成爲 public API 的一部分。如果內部表述在未來版本中改變了（譯註：例如不再使用上頁說的 doubly linked list），`StringList` class 還是需要在輸入（譯註：意指 *deserialized*）時接受 linked-list 型式，並在輸出（譯註：意指 *serialized*）時生成該型式。這個 class 永遠無法擺脫「處理 linked lists」的程式碼 — 即使它不再使用 linked lists。
- 它可能消耗過多空間。本例中的 serialized form 其實不需表現出 linked list 的每一筆記錄（entry）以及所有連結（links）。這些記錄和連結只是實作細節（implementation details），不值得含入 serialized form 內。由於 serialized form 過分龐大，將它寫入磁碟或傳輸於網絡，會耗用過多時間。
- 它可能消耗過多時間。serialization 的整個邏輯並不知道上例的「物件圖拓樸關係」（topology of the object graph），所以它必須經歷一場昂貴的「圖走訪動作」（graph traversal）。本例應該循著 `next` references 進行走訪。
- 它可能造成 stack 滿溢（overflows）。預設的 serialization 程序會對物件圖（object graph）執行遞迴走訪（recursive traversal），那可能會造成 stack 滿溢 — 即使面對的是大小有所節制的物件圖。對「帶有 1200 個元素」的 `StringList` 實體進行 serializing，會在我的機器上引發 stack 滿溢。「造成 stack 滿溢」的元素數量取決於 JVM 實作品。某些 JVM 實作品或許完全不會有這個問題。

`StringList` 的一個合理的 serialized form 是（相對單純的）「list 所含字串個數」加上「字串內容」。這就剝除了 `StringList` 的實際表述細節，構成其邏輯資料表述。下面是 `StringList` 修訂版，內含 `writeObject()`和 `readObject()`，用以實現 serialized form。提醒你，`transient` 飾詞用來表示「某個 *instance* 欄位不出現於 class 的預設 serialized form 中」：

```
// StringList with a reasonable custom serialized form
public class StringList implements Serializable {
   private transient int size   = 0;
   private transient Entry head = null;
```

```
    // No longer Serializable!
    private static class Entry {
        String data;
        Entry  next;
        Entry  previous;
    }

    // Appends the specified string to the list
    public void add(String s) { ... }

    /**
     * Serialize this <tt>StringList</tt> instance.
     *
     * @serialData The size of the list (the number of strings
     * it contains) is emitted (<tt>int</tt>), followed by all of
     * its elements (each a <tt>String</tt>), in the proper
     * sequence.
     */
    private void writeObject(ObjectOutputStream s)
            throws IOException {
        s.defaultWriteObject();
        s.writeInt(size);

        // Write out all elements in the proper order.
        for (Entry e = head; e != null; e = e.next)
            s.writeObject(e.data);
    }

    private void readObject(ObjectInputStream s)
            throws IOException, ClassNotFoundException {
        s.defaultReadObject();
        int size = s.readInt();

        // Read in all elements and insert them in list
        for (int i = 0; i < size; i++)
            add((String)s.readObject());
    }

    ... // Remainder omitted
}
```

注意，即使 `StringList` 的所有欄位都是 `transient`，`writeObject()`還是呼叫 `defaultWriteObject()`，`readObject()`還是呼叫 `defaultReadObject()`。如果所有 *instance* 欄位是 `transient`，技術上允許省略呼叫 `defaultWriteObject()` 和 `defaultReadObject()`，但一般並不推薦如此作爲，因爲即使所有 *instance* 欄位都是 `transient`，喚起 `defaultWriteObject()`還是會影響 serialized form，導致彈性獲得大幅改善。這樣做出來的 serialized form 可使日後新版本有可能添加

nontransient *instance* 欄位而仍舊維護回溯相容和向前相容。這樣的一個物件如果被 *serialized* 於新版本中而被 *deserialized* 於舊版本中，新版本添加的欄位將被忽略。但如果舊版本的 `readObject()`沒有呼叫 `defaultReadObject()`，會造成上述情況下的 deserialization 失敗，並引發 `StreamCorruptedException` 異常。

注意，即使 `writeObject()`是 private 函式，仍有一份文件註解。這種情況類似於 `Name` class 的 private 欄位。這個 private 函式定義了一個 public API：serialized form，而 public API 應該擁有文件！就像`@serial` 標籤之於欄位一樣，`@serialData` 標籤之於函式的意義是：告訴 Javadoc 工具程式將這份說明放到 serialized form 頁面。

讓我們延續稍早的效率討論。如果字串平均長度是 10 個字元，`StringList` 修訂版的 serialized form 佔用的空間大約是舊版的一半。在我的機器上，字串平均長度爲 10 個字元的情況下，`StringList` 修訂版的 *serializing* 速度大約比舊版快 2.5 倍。而且修訂版沒有 stack 滿溢問題，因此 serializable `StringList` 的大小也沒有上限。

預設的 serialized form 對 `StringList` 只是不適任而已，卻可能帶給其他 classes 很糟的結果。對 `StringList` 而言，預設的 serialized form 不夠彈性、效率不佳，但執行結果至少是正確的，*serializing* 和 *deserializing* 之後可獲得原物件的一個忠實複本，帶有完整無缺的約束條件（invariants）。但對於「約束條件與特定之實作細節緊緊相依」的任何物件，就不是這樣。

例如，考慮一個 hash table。其實際表述是一系列 hash buckets，內含一筆一筆的 key-value 記錄（entries）。某筆記錄該被放到哪一個 bucket 去呢？這與 key 的 hash code 有關，而 hash code 在不同的 JVM 實作品中並不保證相同，甚至同一個 JVM 實作品的每次運行也不保證有相同結果。因此 hash table 如果接受預設之 serialized form，會造成嚴重錯誤，這種情況下對 hash table 做 *serializing* 動作和 *deserializing* 動作，可能會導致物件的約束條件受到嚴重破壞。

不論你是否使用預設的 serialized form，每一個 non-`transient` *instance* 欄位將在 `defaultWriteObject()`被喚起時被 *serialized*。因此每一個 *instance* 欄位如果可被標示爲 `transient`，你都應該那麼做。這包括贅餘欄位，亦即其值可根據主要資料欄位（primary data fields）計算而得者，例如 cached hash value；也包括「其值相依於 JVM 某次特定運行」的欄位，例如「用來表現指向 native 資料結構之指標」的一個 `long` 欄位。決定讓某個欄位成爲 nontransient 之前，請確認其值是物件的邏輯狀態（logical state）的一部分。如果你使用自定的 serialized form，大多數或

甚至全部的 *instance* 欄位都應該被標示爲 `transient`，就像先前的 `StringList` 例子那樣。

如果你使用預設的 serialized form，而且你令一個或多個欄位爲 `transient`，記住，這些欄位將在進行 *deserialized* 動作時被初始化爲相應預設值：object reference 欄位的初值爲 null，基本數值欄位的初值爲 0，`boolean` 欄位的初值爲 `false` [JLS, 4.5.5]。如果這些值無法被你的任何 `transient` 欄位接受，你就必須提供一個 `readObject()`，在其中呼叫 `defaultReadObject()`，然後將 `transient` 欄位恢復（設定）爲你可以接受的值（**條款 56**）。另一種做法是，將這些欄位的初始化推遲至它們首次被使用時。

無論你選擇什麼樣的 serialized form，請在你所寫的每一個 serializable class 上宣告一個明確的 serial version UID 欄位。這可避免 serial version UID 成爲潛在的不相容因素（**條款 54**）。這麼做也可獲得微薄的效率利益，因爲如果你沒有提供一個 serial version UID， JVM 就得在執行期間以一個成本高昂的計算爲你生出一個來。

宣告 serial version UID 很簡單，只要爲你的 class 添加以下一行就成了：

```
private static final long serialVersionUID = randomLongValue ;
```

你爲 *randomLongValue* 選擇什麼樣的值，影響並不大。常見的做法是針對你的 class 執行工具程式 `serialver`，便可獲得一個可用數值。任意編造一個值也是可以的。如果你打算爲你的 class 做一個新版本，並且不相容於舊版本，那麼只需改變宣告式中的指定值即可。於是，當有人企圖 *deserialize*「舊版之 *serialized* 實體」時，會失敗並引發 `InvalidClassException` 異常。

簡而言之，當你認定某個 class 應該成爲 serializable（**條款 54**），便應當努力思考使用什麼樣的 serialized form。只有當「預設之 serialized form 是物件邏輯狀態的合理描述」時，才應該使用預設的 serialized form，否則請設計一個自定的 serialized form，用以適切描述物件。你分配給「設計 serialized form」的時間應該和分配給「設計 exported 函式」的時間一樣多。正如你無法在未來新版中消除 exported 函式一樣，你也無法在既有的 serialized form 中消除任何欄位（fields）；它們必須永遠被保存（維護）以確保 serialization 的相容性。因此，如果選擇不良的（甚至錯誤的）serialized form，會對 class 的複雜度和效率帶來永久而負面的衝擊。

條款 56：保護性（防衛性）地寫一個 readObject()

條款 24 有一個用來描述「不可變日期範圍」（*immutable* date-range）的 class，內含可變的（*mutable*）private 日期欄位。這個 class 在建構式和存取函式（accessors）中防衛性地拷貝了 `Date` 物件，藉以極力保存其約束條件（invariants）和不可變性（*immutability*）。下面是這個 class：

```
// Immutable class that uses defensive copying
public final class Period {
    private final Date start;
    private final Date end;

    /**
     * @param  start the beginning of the period.
     * @param  end the end of the period; must not precede start.
     * @throws IllegalArgument if start is after end.
     * @throws NullPointerException if start or end is null.
     */
    public Period(Date start, Date end) {
        this.start = new Date(start.getTime());
        this.end   = new Date(end.getTime());

        if (this.start.compareTo(this.end) > 0)
          throw new IllegalArgumentException(start +" > "+ end);
    }

    public Date start () { return (Date) start.clone(); }

    public Date end ()   { return (Date) end.clone(); }

    public String toString() { return start + " - " + end; }

    ... // Remainder omitted
}
```

假設你決定讓這個 class 成爲 serializable。由於一個 `Period` 物件之實際表述精確反映出其邏輯資料內容，因此，本例使用預設的 serialized form 並無不合理（**條款 55**）。爲了讓這個 class 成爲 serializable，也許你惟一需要做的事情就是在 class 宣告式中加上 "`implements Serializable`"。然而如果你這麼做了，這個 class 將不再保證其關鍵的約束條件（invariants）。

問題在於，`readObject()`可被視爲另一種富有效力的 public 建構式，你必須像對待任何建構式一樣小心謹慎地對待它。就像建構式必須檢查其引數有效性（**條款 23**）並適當製作參數的防衛性複本（defensive copies）（**條款 24**）一樣，`readObject()`

也必須如此。如果 `readObject()`沒有能夠成功完成上述每一件事，就會導致「蓄意攻擊者」可以輕鬆違反 class 的約束條件（invariants）。

寬鬆地說，`readObject()`像個建構式，接受一組 byte stream 做爲惟一參數。正常情況下 byte stream 的成因是「將一個正常建構而得的實體 *serializing*」而得。但是當 `readObject()`面對一組人工製造的 byte stream，而後者用來刻意產生一個「違反 class 約束條件」之物件時，問題就爆發了。假設我們僅僅爲 `Period` 的宣告式添加 `"implements Serializable"`，下面這個醜陋的程式會生成一個 `Period` 實體，其結束日期竟在起始日期之前（譯註：違反 `Period` 的約束條件）：

```
public class BogusPeriod {
    // Byte stream could not have come from real Period instance
    private static final byte[] serializedForm = new byte[] {
    (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x06,
    0x50, 0x65, 0x72, 0x69, 0x6f, 0x64, 0x40, 0x7e, (byte)0xf8,
    0x2b, 0x4f, 0x46, (byte)0xc0, (byte)0xf4, 0x02, 0x00, 0x02,
    0x4c, 0x00, 0x03, 0x65, 0x6e, 0x64, 0x74, 0x00, 0x10, 0x4c,
    0x6a, 0x61, 0x76, 0x61, 0x2f, 0x75, 0x74, 0x69, 0x6c, 0x2f,
    0x44, 0x61, 0x74, 0x65, 0x3b, 0x4c, 0x00, 0x05, 0x73, 0x74,
    0x61, 0x72, 0x74, 0x71, 0x00, 0x7e, 0x00, 0x01, 0x78, 0x70,
    0x73, 0x72, 0x00, 0x0e, 0x6a, 0x61, 0x76, 0x61, 0x2e, 0x75,
    0x74, 0x69, 0x6c, 0x2e, 0x44, 0x61, 0x74, 0x65, 0x68, 0x6a,
    (byte)0x81, 0x01, 0x4b, 0x59, 0x74, 0x19, 0x03, 0x00, 0x00,
    0x78, 0x70, 0x77, 0x08, 0x00, 0x00, 0x00, 0x66, (byte)0xdf,
    0x6e, 0x1e, 0x00, 0x78, 0x73, 0x71, 0x00, 0x7e, 0x00, 0x03,
    0x77, 0x08, 0x00, 0x00, 0x00, (byte)0xd5, 0x17, 0x69, 0x22,
    0x00, 0x78 };

    public static void main(String[] args) {
        Period p = (Period) deserialize(serializedForm);
        System.out.println(p);
    }

    // Returns the object with the specified serialized form
    public static Object deserialize(byte[] sf) {
        try {
            InputStream is = new ByteArrayInputStream(sf);
            ObjectInputStream ois = new ObjectInputStream(is);
            return ois.readObject();
        } catch (Exception e) {
            throw new IllegalArgumentException(e.toString());
        }
    }
}
```

這份用來初始化 `serializedForm` 的 `byte` array 字面常數如何誕生的呢？它的產生過程是先*serializing*一個正常的`Period`實體，然後手工編輯輸出結果。這個 stream 的細節並不重要，但如果你很好奇，serialization byte stream 的格式描述於 *Java*™ *Object Serialization Specification* [Serialization, 6]。執行上述程式，會列印出 "Fri Jan 01 12:00:00 PST 1999 - Sun Jan 01 12:00:00 PST 1984."（譯註：起始日期晚於終止日期）。啊呀，讓 `Period` 成爲 serializable 竟使我們得以創建一個「違反 `Period` class 約束條件」的物件。欲修正這個問題，我們應該爲 `Period` 提供一個 `readObject()`，在其中呼叫 `defaultReadObject()`然後檢查 *deserialized* 物件的有效性；如果檢驗不通過就拋出 `InvalidObjectException`，阻止 deserialization 動作：

```
private void readObject(ObjectInputStream s)
    throws IOException, ClassNotFoundException {
    s.defaultReadObject();

    // Check that our invariants are satisfied
    if (start.compareTo(end) > 0)
        throw new InvalidObjectException(start +" after "+ end);
}
```

雖然這份修正可以阻止「蓄意攻擊者」創建一個無效的（違反約束條件的）`Period` 實體，其中卻還潛伏一個更隱微的問題：允許攻擊者創建一個可變的（*mutable*）`Period` 實體，作法是杜撰一組 byte stream，其中先以一組 byte stream 表現一個有效的 `Period` 實體，然後是額外的 references 指向 `Period` 實體內的 private `Date` 欄位。「蓄意攻擊者」首先從 `ObjectInputStream` 讀取 `Period` 實體，再讀取其後「詐欺性的 object references」。這些 references 使攻擊者得以存取 `Period` 物件中的 private `Date` 欄位所指涉（指向、引用）的物件。只要改變那些 `Date` 實體，攻擊者也就改變了 `Period` 實體。以下 class 可以示範這一類攻擊：

```
public class MutablePeriod {
    // A period instance
    public final Period period;

    // period's start field, to which we shouldn't have access
    public final Date start;

    // period's end field, to which we shouldn't have access
    public final Date end;
```

```
    public MutablePeriod() {
        try {
            ByteArrayOutputStream bos =
                new ByteArrayOutputStream();
            ObjectOutputStream out =
                new ObjectOutputStream(bos);

            // Serialize a valid Period instance
            out.writeObject(new Period(new Date(), new Date()));

            /*
             * 添加詐欺性的 "previous object refs"，欺詐對象是
             * Period 內的 Date 欄位。細節請見 "Java Object Serialization
             * Specification," Section 6.4.
             */
            //譯註：從本程式碼絕對看不出所以然來，必須參考上述規格書 6.4 節，
            //      該節談的是十分細瑣的"Grammar for the Stream Format"
            byte[] ref = { 0x71, 0, 0x7e, 0, 5 }; // Ref #5
            bos.write(ref); // The start field
            ref[4] = 4;     // Ref # 4
            bos.write(ref); // The end field

            // Deserialize Period and "stolen" Date references
            ObjectInputStream in = new ObjectInputStream(
                      new ByteArrayInputStream(bos.toByteArray()));
            period = (Period) in.readObject();
            start  = (Date)   in.readObject();
            end    = (Date)   in.readObject();
        } catch (Exception e) {
            throw new RuntimeException(e.toString());
        }
    }
}
```

如果你想看看實際攻擊行動，請執行以下程式：

```
public static void main(String[] args) {
    MutablePeriod mp = new MutablePeriod();
    Period p = mp.period;
    Date pEnd = mp.end;

    // 讓我們倒轉時鐘
    pEnd.setYear(78);
    System.out.println(p);

    // 再倒回 60 年代!
    pEnd.setYear(69);
    System.out.println(p);
}
```

執行這個程式後，產生以下輸出：

```
Wed Mar 07 23:30:01 PST 2001 - Tue Mar 07 23:30:01 PST 1978
Wed Mar 07 23:30:01 PST 2001 - Fri Mar 07 23:30:01 PST 1969
```

如此一來，雖然 Period 實體被創建時其約束條件（invariants）未遭破壞，但任意修改其內部成分還是可能的。一旦攻擊者擁有一個可變的（*mutable*）Period 實體，他就可以引發巨大的傷害：他可以把這個實體傳給一個「安全性（security）倚賴 Period 的不變性（*immutability*）」的一個 class。這並不牽強，有些 classes 的安全性真的取決於 String 的不變性。

問題的根源在於，Period 的 readObject() 並沒有做到足夠的保護性拷貝（defensive copying）。當一個物件被 *deserialized* 時，很重要的一件事是，任何欄位如果內含「客戶端絕對不得擁有」的 object reference，就該對那些欄位進行保護性拷貝。因此每一個 serializable *immutable* class 如果內含 private *mutable* 成分，都必須在其 readObject() 中保護性地拷貝這些成分。下面的 readObject() 足以確保 Period 的約束條件並維護 Period 的不變性（*immutability*）：

```
private void readObject(ObjectInputStream s)
    throws IOException, ClassNotFoundException {
    s.defaultReadObject();

    // Defensively copy our mutable components
    start = new Date(start.getTime());
    end   = new Date(end.getTime());

    // Check that our invariants are satisfied
    if (start.compareTo(end) > 0)
        throw new InvalidObjectException(start +" after "+ end);
}
```

注意，保護性拷貝在有效性檢驗（validity check）之前執行，而且我們並不使用 Date clone() 來完成。這兩個細節對於保護 Period 免受攻擊（**條款 24**）是必要的。也請注意，保護性拷貝不可能針對 final 欄位進行，因此爲了使用 readObject()，我們必須令 start 和 end 兩欄位爲 nonfinal。這真是不幸，但明顯是相對較好的辦法。有了這個新的 readObject() 並移除 start 和 end 欄位身上的 final 飾詞後，MutablePeriod class 不再有效（也就失去其攻擊性）。先前的攻擊程式如今產生以下輸出：

```
Thu Mar 08 00:03:45 PST 2001 - Thu Mar 08 00:03:45 PST 2001
Thu Mar 08 00:03:45 PST 2001 - Thu Mar 08 00:03:45 PST 2001
```

有一個簡單的「石蕊試劑」可以決定預設的 `readObject()`是否可被接受。你對於以下動作是否覺得舒服：「增加一個 public 建構式，其參數對應於你的物件的每一個 nontransient 欄位，並且不做有效性檢驗就將那些值儲存於對應欄位中？」如果你回答 yes，那麼你必須明確提供一個（而非使用預設的）`readObject()`，它必須執行建構式所需要的每一個有效性檢驗和保護性拷貝。

對於 nonfinal serializable classes，`readObject()`和建構式之間還存在另一層相似性：`readObject()`不得直接或間接喚起一個「可被覆寫」（overridable）的函式（**條款 15**）。如果你違反這條規則，而且你所呼叫的那個「可被覆寫的」函式真的被覆寫了，那麼該函式將執行於「subclass 狀態（state, 亦即欄位）被 *deserialized*」之前。這很可能導致程式失敗。

總而言之，任何時候當你寫下一個 `readObject()`，請想像你正寫下一個 public 建構式，無論收到什麼樣的 byte stream，它都必須生產出一個有效實體。不要假設那個 byte stream 一定代表一個「真正被 *serialized* 的實體」（譯註：意思是可能有僞造的）。雖然本條款關心的是「使用預設 serialized form」的 class，但所有的討論也都適用於「使用自定 serialized forms」的 class。下面總結出撰寫「防彈式」`readObject()`的一些準則：

- 如果 class 帶有 object reference 欄位，而且後者必須保持爲 private，請對每一個被儲存於如此欄位的物件進行保護性拷貝。*immutable* classes 的 *mutable*（可變）成分就屬此類。
- 面對帶有約束條件（invariants）的 class，應檢驗其約束條件並於檢驗失敗時拋出 `InvalidObjectException` 異常。檢驗動作應該緊跟在任何保護性拷貝之後。
- 如果整個物件圖（object graph）在被 *deserialized* 之後必須確認有效，那就應該使用 `ObjectInputValidation` interface。不過這個 interface 的使用已超越本書設定範圍。你可以在《*The Java Class Libraries, Second Edition, Volume 1*》[Chan98, p1256] 找到一個實例。
- 無論直接或間接，不要在 `readObject()`中喚起 class 內可被覆寫（overridable）的任何函式。

`readResolve()`或許可用來替換上述具有保護功能的 `readObject()`。**條款 57** 將討論這個主題。

條款 57：必要時提供一個 readResolve()

條款 2 描述 ***Singleton*** 範式（pattern）並提出以下例子。這個 class 將限制其建構式被使用，確保只可能有一個實體被創建出來：

```
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();

    private Elvis() {
        ...
    }

    ... // Remainder omitted
}
```

就如**條款 2** 所說，如果在上述 class 的宣告式添加 "`implements Serializable`" 字樣，它將不再是個 ***singleton*** — 無論這個 class 使用預設的或自定的 serialized form（**條款 55**），也無論這個 class 是否提供一個明確的 `readObject()`（**條款 56**）。要知道，任何 `readObject()`，不論明確寫出的版本或預設的版本，都應該傳回一個新被創建的實體，此實體不同於「class 初始化時期被創建」的實體（譯註：class 初始化時期意指 "static"）。在 JDK 1.2 發表之前，我們不可能寫出一個 serializable ***singleton*** class。

JDK 1.2 爲 serialization 添加了 `readResolve()`特性 [Serialization, 3.6]。如果一個即將被 *deserialized* 的物件，其 class 定義了一個 `readResolve()`，並有著恰當的宣告，這個函式將在 *deserialized* 之後被新建物件喚起，而它所傳回的 object reference 將被用來替代新建物件。這一性質的大多數運用實例中並沒有留住「指向新建物件」的 reference，該物件實際上流產了，立刻成爲垃圾回收機制的合格候選人。

如果 `Elvis` class 打算實現 `Serializable`，下面的 `readResolve()`足夠保證其 ***singleton*** 性質：

```
private Object readResolve() throws ObjectStreamException {
    // Return the one true Elvis and let the garbage collector
    // take care of the Elvis impersonator.
    return INSTANCE;  //譯註：這個 INSTANCE 是獨一無二的 static final，見本頁上。
}
```

這個函式略掉 *deserialized* 物件，簡單傳回「class 初始化時期所創建的那個 `Elvis` 實體」。因此 `Elvis` 實體的 serialized form 不需內含任何真實資料；所有 *instance* 欄位都應該標示爲 `transient`。這種做法不僅適合 `Elvis`，也適合所有 ***singleton***s。

`readResolve()`不只對 ***singleton***s 有其必要性，對其他所謂「實體個數受管控」（instance-controlled）— 亦即「嚴格控制實體創建動作以維護某種約束條件（invariant）」— 的 classes 也很重要。「實體個數受管控」的另一個例子是 typesafe enum（**條款 21**），其 `readResolve()`必須傳回「用以表現特定 enum 常數」之正典實體（canonical instance）。通常如果你撰寫一個 serializable class 而它並不內含任何 public 或 protected 建構式，請考慮它也許需要一個 `readResolve()`。

`readResolve()`的第二個用途是做爲保護性的（defensive）`readObject()`（如**條款 56** 所推薦）的一個保守替代選擇。在此種用法中，`readObject()`內的所有有效性檢驗和保護性拷貝都消失了，改用一般建構式提供的有效性檢驗和保護性拷貝。如果用的是預設的 serialized form，那麼 `readObject()`甚至可完全取消。就像**條款 56** 所言，這使不懷好意的客戶得以創建一個危及約束條件的實體。然而這個潛在受損的 *deserialized* 實體絕不會進入「運作中的服務」（active service）內，它只會被開採出來而後輸入到一個 public 建構式或 static ***factory*** 中，然後就被丟棄。

這個作法的漂亮在於，它實際上差不多消除了 serialization 機制之中「語言學以外」的成分，使任何 class 都不可能違反「成爲 serializable 之前」所存在的任何約束條件。爲了更具體說明這項技術，下面的`readResolve()`可用來替代**條款 56** 的`Period` class 的保護性（defensive）`readObject()`：

```
// The defensive readResolve idiom
private Object readResolve() throws ObjectStreamException {
    return new Period(start, end);
}
```

上述的 `readResolve()`可以阻擋**條款 56** 描述的兩種攻擊手法。保護性的`readResolve()`和保護性的 `readObject()`相比，有數個優點：(1) 這是一種十分機械化的技術，可令一個 class 成爲 serializable 而必不至於危害其約束條件。(2) 只需要少量程式碼和一些些思考，保證可以有效運作。(3) 消除了「serialization 用於 final 欄位」時的人爲束縛。

雖然保護性 `readResolve()`技法並未被廣泛採用，但它其實很值得考慮。它的主要缺點是：不適合那些「可在 package 之外被繼承」的 classes。這對 *immutable* classes 沒有影響，因爲它們通常是 final（**條款 13**）。此技法的一個次要缺點是：會輕微降低 deserialization 的效率，因爲需要創建一個額外物件。在我的機器上，和保護性 `readObject()`相比，`readResolve()`會降低 `Period` 實體的 deserialization 效率大約一個百分點。

`readResolve()`的可存取性（accessibility）很重要。如果你在 final class（例如一個 ***singleton***）中放置 `readResolve()`，應該把它設爲 private。如果你在 nonfinal class 中放置 `readResolve()`，必須謹慎考慮其可存取性。如果讓它成爲 private，將無法施行於任何 subclasses 身上。如果讓它成爲 package-private，只能施行於同一個 package 內的 subclasses。如果讓它成爲 protected 或 public，可施行於所有並未對它進行覆寫（*override*）的 subclasses 身上。但是請注意，如果 `readResolve()`在 superclass `B` 中是 protected 或 public，而 subclass `D` 並未覆寫它，那麼 *deserializing* 一個「*serialized* `D` 實體」將會產出一個 `B` 實體，那或許不是你所想要的。

上一段文字暗示，在「允許被繼承」的 class 中，`readResolve()`可能並不適合取代保護性的 `readObject()`。因爲如果 superclass 的 `readResolve()`是 final（譯註：不一定是 final，只要未被覆寫都算），將會妨礙 subclass 實體被適當地 *deserialized*。但如果 `readResolve()`被覆寫，不懷好意的 subclass 卻又可以把它覆寫爲「傳回一個受損實體」的函式。

綜合言之，你必須使用 `readResolve()`來保護 ***singleton***s 或其他「對實體個數有所管控的」（instance-controlled）classes 的「實體個數約束條件」。本質而言，`readResolve()`改變了 `readObject()`，從一個「實際效用上的 public 建構式」改變爲一個「實際效用上的 public static ***factory***（靜態工廠）」。對於「package 之外禁止繼承」的 classes 而言，`readResolve()`也可被用來做爲保護性 `readObject()`的一個簡單替代方案。

參考書目

Reference

[Arnold00] Arnold, Ken, James Gosling, David Holmes. *The Java™ Programming Language, Third Edition*. Addison-Wesley, Boston, 2000. ISBN: 0201704331.

[Beck99] Beck, Kent. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 1999. ISBN: 0201616416.

[Bloch99] Bloch, Joshua. Collections. In *The Java™ Tutorial Continued: The Rest of the JDK™*. Mary Campione, Kathy Walrath, Alison Huml, and the Tutorial Team. Addison-Wesley, Reading, MA, 1999. ISBN: 0201485583. Pages 17-93. Also available as <http://java.sun.com/docs/books/tutorial/collections/index.html>.

[Campione00] Campione, Mary, Kathy Walrath, Alison Huml. *The Java™ Tutorial Continued: A Short Course on the Basics*. Addison-Wesley, Boston, MA, 2000. ISBN: 0201703939. Also available as <http://java.sun.com/docs/books/tutorial/index.html>.

[Cargill96] Cargill, Thomas. Specific Notification for Java Thread Synchronization. *Proceedings of the Pattern Languages of Programming Conference*, 1996.

[Chan00] Chan, Patrick. *The Java™ Developers Almanac 2000*, Addison-Wesley, Boston, MA, 2000. ISBN: 0201432994.

[Chan98] Chan, Patrick, Rosanna Lee, and Douglas Kramer. *The Java™ Class Libraries Second Edition, Volume 1*, Addison-Wesley, Reading, MA, 1998. ISBN: 0201310023.

[Collections] *The Collections Framework.* Sun Microsystems. March 2001. <http://java.sun.com/j2se/1.3/docs/guide/collections/index.html>.

[DocCheck] *Doc Check.* Sun Microsystem. October 2001. <http://java.sun.com/j2se/javadoc/doccheck/index.html>.

[Flanagan99] Flanagan, David. *Java™ in a Nutshell, Third Edition*, O'Reilly and Associates, Sebastopol, CA, 1999. ISBN: 1565924878.

[Gamma95] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995. ISBN: 0201633612.

[Gong99] Gong, Li. *Inside Java™ 2 Platform Security*, Addison-Wesley, Reading, MA, 1999. ISBN: 0201310007.

[Heydon99] Allan Heydon and Marc A. Najork. Performance Limitations of the Java Core Libraries. In *ACM 1999 Java Grande Conference*, pages 35-41. ACM Press, June 1999. Also available as <http://research.compaq.com/SRC/mercator/papers/Java99/final.pdf>

[Horstman00] Horstmann, Cay, and Gary Cornell. *Core Java™ 2: Volume II Advanced Features*, Prentice Hall, Palo Alto, CA, 2000. ISBN: 0130819344.

[HTML401] *HTML 4.01 Specification*. World Wide Web Consortium. December 1999. <http://www.w3.org/TR/1999/REC-html401-19991224/>.

[J2SE-APIs] *Java™ 2 Platform, Standard Edition, v 1.3 API Specification*. Sun Microsystems. March 2001. <http://java.sun.com/j2se/1.3/docs/api/overview-summary.html>.

[Jackson75] Jackson, M.A. *Principles of Program Design*, Academic Press, London, 1975. ISBN: 0123790506.

[JavaBeans] *JavaBeans™ Spec*. Sun Microsystems. March 2001. <http://java.sun.com/products/javabeans/docs/spec.html>.

[Javadoc-a] *How to Write Doc Comments for Javadoc*. Sun Microsystems. January 2001. <http://java.sun.com/j2se/javadoc/writingdoccomments/>.

[Javadoc-b] *Javadoc Tool Home Page*. Sun Microsystems. January, 2001. <http://java.sun.com/j2se/javadoc/index.html>.

[JLS] Gosling, James, Bill Joy, Guy Steele, Gilad Bracha. *The Java™ Language Specification, Second Edition*, Addison-Wesley, Boston, 2000. ISBN: 0201310082.

[Kahan91] Kahan, William, and J. W. Thomas. *Augmenting a Programming Language with Complex Arithmetic*, UCB/CSD-91-667, University of California, Berkeley, 1991.

[Knuth74] Knuth, Donald. Structured Programming with go to Statements. *Computing Surveys* 6 (1974): 261-301.

[Lea01] *Overview of Package util.concurrent Release 1.3.0*. State University of New York, Oswego. January 12, 2001. <http://g.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html>.

[Lea00] Lea, Doug. *Concurrent Programming in Java™: Design Principles and Patterns, Second Edition*, Addison-Wesley, Boston, 2000. ISBN: 0201310090.

[Lieberman86] Lieberman, Henry. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 214-223, Portland, September 1986. ACM Press.

[Meyers98] Meyers, Scott. *Effective C++, Second Edition: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, Reading, MA, 1998. ISBN: 0201924889.

[Parnas72] Parnas, D.L. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM* 15 (1972): 1053-1058.

[Posix] 9945-1:1996 (ISO/IEC) [IEEE/ANSI Std. 1003.1 1995 Edition] Information Technology Portable Operating System Interface (POSIX) — Part 1: System Application: Program Interface (API) [C Language] (ANSI), IEEE Standards Press, ISBN: 1559375736.

[Pugh01a] *The Java Memory Model*. Ed. William Pugh. University of Maryland. March 2001. <http://www.cs.umd.edu/~pugh/java/memoryModel/>.

[Pugh01b] *The Double-Checked Locking is Broken™ Declaration*. Ed. William Pugh. University of Maryland. March 2001. <http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>.

[Serialization] *Java™ Object Serialization Specification*. Sun Microsystems. March 2001. <http://java.sun.com/j2se/1.3/docs/guide/serialization/spec/serialTOC.doc.html>.

[Smith62] Smith, Robert. Algorithm 116 Complex Division. *In Communications of the ACM*, 5.8 (August 1962): 435.

[Snyder86] Synder, Alan. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, 38-45, 1986. ACM Press.

[Thomas94] Thomas, Jim, and Jerome T. Coonen. Issues Regarding Imaginary Types for C and C++. In *The Journal of C Language Translation*, 5.3 (March 1994): 134-138.

[Vermeulen00] Vermeulen, Allan, Scott W. Ambler, Greg Bumgardener, Eldon Metz, Trevor Mesfeldt, Jim Shur, Patrick Thompson. *The Elements of Java™ Style*, Cambridge University Press, Cambridge, United Kingdom, 2001. ISBN: 0521777682.

[Wulf72] Wulf, W. A Case Against the GOTO. *Proceedings of the 25th ACM National Conference* 2 (1972): 791-797.