實戰 *Java*

— 九個別具特色的實作經驗 —

(**The Art of Java**, by Herbert Schildt & James Holmes)

— 侯 捷　譯 —

出版序

《實戰 *Java* — 十個別具特色的實作經驗》與目前市面上眾多 Java 書籍的最大不同，在於本書既非基礎觀念之教學書籍，亦非開發工具之使用手冊，而是以「一章一專案」的方式，從實際應用面引領讀者領略 Java。

本書是《*The Art of Java*》的中文譯本。原作者 Herbert Schildt 是一位學有專精、著作等身的 IT 技術作家，其作品普遍獲得良好評價。另一位作者 James Holmes 專注於 Java 相關技術研究，成就獲得業界人士推崇。

本書是《*The Art of Java*》的新譯本，此前譯本名為《大師談 Java》。承侯捷先生慨然相助，接下重新翻譯的工作，在此特別致謝。侯捷先生對於譯作的嚴謹態度，向為兩岸三地讀者肯定。但盼好書加上優質譯者，能夠為讀者們帶來最好的學習經驗。

Herbert Schildt 另著有姊妹作《*The Art of C++*》，其中譯本《實戰 *C++* — 八個別具特色的實作經驗》亦由侯捷先生翻譯。

上奇科技出版事業處編輯部

侯捷譯序

市面上充滿了程式語言基本語法書籍、編程工具使用手冊、手把手蝸步緩進的「傻瓜系列」[1]。就像人生在不同層次有不同的體會，這些書籍在你不同的學習過程中也必然各有貢獻。但是當我們上升到某個檔次，驟然發現，高處不勝寒，無書可讀了！

倒不是說我們層次多高，高到無書可讀，而是書市中面向大眾的稍稍高檔次書籍 — 尤其在程式語言及更高階領域 — 很少。做為一個「對業內技術及程式員保持密切觀察」的讀者，我知道這種「稀缺」其實不能反映需求。做為一個技術書籍作者，我也知道，這種「稀缺」其實反映出陽春白雪的必然性。

學習一門語言，基本語法永遠不是問題，也不成其為關鍵，關鍵在於如何適當地運用。運用是最好的練習；實踐是檢驗力量的最佳辦法。這本《實戰 *Java* — 1 個別具特色的實作經驗》，就是很好的實踐藍本。您不但可以從書中獲得中大型程式的組織和寫作示範，還可以獲得九個不同領域（解析器 *Parser*、直譯器 *Interpreter*、檔案下載、E-mail 服務、網頁爬行、HTML 瀏覽、統計與繪圖、金融計算、人工智慧搜尋）的領域知識。最終，您還獲得了九個可以做為個人開發起點的良好基礎。

本書的行進方式是，以程式碼為主軸，輔以大量說明。對於已經脫離語法學習階段，希望獲得實作經驗的讀者，這種行進方式頗為合適。程式碼很多，作者又不厭其煩地整體性條列、局部性條列，必會有些讀者嫌其囉嗦佔篇幅。但如果反省「買書當成買紙」的不當心態，從而不再以區區紙頁的角度去衡量一本書（的方方面面），而是從整體知識、便利閱讀、節省時間的角度來看，我相信詳盡的解說永遠受歡迎。

[1] 這裡並無不敬之意。與「傻瓜相機」用法同。

在中英術語的採納上，本書視情況時而使用中文，時而使用英文[2]，時而中英並陳。已做悉心安排，並多次檢閱試驗，必不致令讀者混亂困惑。本書目錄末尾列有中英對照表，每章第一頁亦列有該章主要術語的中英對照。

一如「出版序」所言，本書另有姊妹作：《實戰 *C++* — 八個別具特色的實作經驗》，亦由上奇出版。對於以 C++為工具的程式員而言，又是一個好選擇。

侯捷 2005/04/20 於臺灣新竹
jjhou@jjhou.com
http://www.jjhou.com（繁體） http://jjhou.csdn.net（簡體）

[2] 使用英文術語的兩大用意：(1) 某些術語業界習慣以英文稱呼，(2) 某些術語譯為中文頗覺彆扭，例如 Java "method" 譯為 "方法"，雖四平八穩但讀之不快，又不具術語突出性。當然，這是譯者的個人取向及其業界經驗和教學經驗使然，您也可能有不同看法。

目 錄

本書術語中英對照

計算機術語何其浩繁。下表只列本書中犖犖大者。各章第一頁另有該章術語摘要整理。

英文術語	中文術語	英文術語	中文術語
abstract	抽象的	access	存取、取用
adapter	配接器	address	位址
algorithm	演算法	argument	引數
array	陣列	assignment	指派、賦值
A.I.	人工智慧	batch	批次（整批作業）
cache	快取（區）	check box	核取方塊
class	類別	client	客端、客戶端、客戶
client-server	主從架構	collection	群集
combo box	複合方塊、複合框	command line	命令列
compiler	編譯器	component	組件（用於 GUI 則譯為控件）
concurrent	並行	connection	連接、連線
control	控件	constructor	建構式
escape code	轉義碼	evaluate	評估、求值、核算
event	事件	exception	異常
expression	算式、表達式	file	檔案
flag	旗標	function	函式
GUI	圖形介面	handle	識別碼
header file	表頭檔	import	匯入
instance	實體	instantiate	具現化
interface	介面（編程）、界面（GUI）	Internet	網際網路
interpreter	直譯器	invoke	喚起
label	標籤	library	程式庫
link	連結	list	串列、清單、系列
macro	巨集	memory	記憶體
menu	選單	method (in Java)	函式
object	物件	overload	重載
parameter	參數	parse	解析
polymorphism	多型	process	行程
programming	編程、程式設計	progress bar	進度條
regular expression	正則表達式	source	源碼
statement	述句	stream	資料流、串流
tag	頁籤	thread	緒、緒程
token	語彙單元	type	型別
variable	變數	WWW	萬維網

關於作者

Herbert Schildt 是 C, C++, Java 和 C# 等語言的領導專家，亦擅長編寫 Windows 程式。他的編程書籍全球銷量超過 300 萬冊，並被翻譯成所有主要語言。他是多部 C++ 暢銷書的作者，包括《*C++: The Complete Reference*》,《*C++ From the Ground Up*》,《*C++: A Beginner's Guide*》, 以及《*STL Programming From the Ground Up*》。他的其他暢銷書包括：《*C: The Complete Reference*》,《*Java 2:The Complete Reference*》和《*C#: The Complete Reference*》。Schildt 獲得伊利諾州立大學計算機科學碩士學位。您可透過其諮詢辦公室電話（217）586-4683 與他聯繫。

James Holmes 是一位在 Java 編程領域受到表彰的領導者。他獲得 2002 Oracle Magazine 的 Java 年度開發者稱謂，並且是 Jakarta Structs 源碼開放專案的 Committer。他目前是一位獨立的 Java 顧問，也是 Sun 認證之 Java 程式員及網站組件（Web Component）開發人員。您可透過電子郵箱 james@jamesholmes.com 和他聯絡，也可以訪問他的網站 http://www.JamesHolmes.com。

關於譯者

侯捷 對於 C, C++, Java 和 C# 等語言皆有研究，亦擅長編寫 Windows 程式。他的各種著作和譯作在兩岸三地擁有 50 萬以上讀者。他的主要著作包括《深入淺出 *MFC*》、《多型與虛擬》、《*STL* 源碼剖析》、《無責任書評》卷 1,2,3、《*Word* 排版藝術》、《左手程式右手詩》,譯作包括:《*C++Primer*》、《*Effective C++*》、《*More Effective C++*》、《*The C++ Standard Library*》、《*Generic Programming and the STL*》、《*Refactoring*》、《*System Programming in Windows 95*》和《*Windows 95 System Programming Secrets*》等十數本。侯捷目前在元智大學、南京大學、同濟大學授課。您可透過電子郵箱 jjhou@jjhou.com 與他聯繫，也可以訪問他的個人網站 http://www.jjhou.com（中文繁體）和 http://jjhou.csdn.net（中文簡體）。

0

作者序

By Herb Schildt

自 1991 年起，Sun Microsystem 公司的 James Gosling, Patrick Naughton, Chris Warth, Ed Frank 和 Mike Sheridan，開始著手一個最終撼動整個編程界的新型程式語言。這個一開始名為 "Oak" 的語言於 1995 年被改名為 "Java" －－ 這在電腦界也是一件前所未見的事。

Java 在兩個重要方向上改變了編程路線。首先，Java 結合的特色可促進 Internet 應用程式的開發。Java 可說是世界上第一個真正準備好在 Internet 上大顯身手的語言。其次，Java 在電腦語言設計上跨了一大步，例如它重新定義了物件範型（object paradigm）、簡化了異常（exception）處理、把多緒（multithreading）徹底整合到語言中，而且能夠產出名為 bytecode 的一種可攜式目的碼（portable object code），這種技術使程式得以在各種異質平台上運作。

因此，可以說，Java 對資訊界的重要性奠定在兩大支柱上：內建的 Internet 支援能力，以及電腦語言設計上的提升。任何一個原因便足以使 Java 成為好語言，兩個原因加起來更使 Java 成為偉大的程式語言，確立了它在電腦史上的地位。

本書將向您展示，為什麼 Java 是如此獨特非凡的一個語言。

本書有些什麼

本書和大多數 Java 書籍相當不同。其他許多書籍教授語言基礎，本書卻是告訴您如何把 Java 應用到一些更有趣、更有用，有時候甚至不可思議的的編程工作上。過程中將展現 Java 語言的威力、多功能，和優雅。唯有透過 Java 運用上的藝術，才能將 Java 設計上的藝術性展露無遺。

一如您所預期，書中會出現一些直接和 Internet 有關的應用程式，例如第 4 章的檔案下載器，或第 5 章的 E-mail 客戶端程式。然而，另有許多章節的程式碼，向您展示 Java 在 Internet 以外的豐富性。例如第 3 章的語言直譯器，或第 10 章以人工智慧為基礎的搜尋法（AI-based search

routines），就是我們所謂的「純淨碼」（pure code）。這兩個應用程式都不倚賴 Internet 或用到 GUI 介面，它們都是以往人們可能會預期以 C++ 寫出來的類型。以 Java 輕鬆開發出這些類型的程式，正足以展現 Java 的多功能和靈活性。

每一章開發出來的程式碼，都可以在無任何修改的情況下直接使用。例如第 2 章的算式解析器（expression parser）就有可能完美附加到許多應用程式身上。然而當您把這些程式拿來當作自己開發工作的出發點，最大的好處才真正顯現。例如第 6 章的網頁爬行器（Web crawler）可被改寫成網站歸檔器（Web-site archiver）或中斷連結偵測器（broken-link detector）。不妨把這些程式和次系統當作您的專案的一個初期發展平台。

假設您有這些 Java 知識

本書假設您在 Java 語言上有堅實的基礎。您應該能夠自行產生、編譯、執行 Java 程式。您應該能夠運用最一般性的 Java API、處理異常，並寫出一個多緒程式。本書假設您已經擁有 Java 初階課程中應該獲得的能力。

如果需要複習或加強基本知識，我推薦以下兩本書：

- *Java 2: A Beginner's Guide*
- *Java 2: The Complete Reference*

這兩本書都已經由 McGraw-Hill/Osborne 出版。

共同合作的收穫

我撰寫編程方面的書籍已有多年，很少與人共同執筆。本書是個例外。由於一些出乎意料但令人快樂的事情發生，使我得以和電腦界最閃耀的明星之一 James Holmes 合作。James 是一位相當傑出的程式員，擁有不少令人敬佩的成就，包括 Orcale「年度 Java 開發者」，以及 Jakarta Struts 專案的交付委員（committer）。由於 James 在 Web-based 編程工作上擁有獨到見解，我認為如果他可以為本書貢獻一些篇章的話，那真是太好了。幸運的是我竟然能夠說服他一同參與。最後 James 寫下了 4、5、6、7 共四章，內含最精深透徹的 Internet 應用程式。他對本書帶來莫大的貢獻。

James 目前正進行一本關於 Struts 方面的深入書籍：《*Struts: The Complete Reference*》，這本書將在 2003 年底問世。

別忘了：程式碼就在網上

請記住，本書所有範例和專案的原始碼，都可以在 www.osborne.com 網站上免費取得。

Herb Schildt 的其他作品

本書只是 Herb Schildt 編程系列書籍中的一冊。下面是您可能也感興趣的其他書籍。

如果想學習更多的 Java 編程，我們推薦以下書籍：

- *Java 2: The Complete Reference*
- *Java 2: A Beginner's Guide*
- *Java 2: Programmer's Reference*

如果想學習 C++，您會發現下面這些書特別有用：

- *C++: The Complete Reference*
- *C++: A Beginner's Guide*
- *Teach Yourself C++*
- *C++ From the Ground Up*
- *STL Programming From the Ground Up*

如果想學習 C#，我們建議以下書籍：

- *C#: A Beginner's Guide*
- *C#: The Complete Reference*

如果想學習更多 C 語言 — 現代編程技術的基礎，下面這些書頗有趣味：

- *C: The Complete Reference*
- *Teach Yourself C*

James Holmes 的其他作品

如果您想學習 Struts—一個用以進行 Web 開發的源碼開放應用框架，我們推薦以下的 James Holmes 作品：

- *Struts: The Complete Reference*

1

1. Java原理

The Genius of Java

By Herb Schildt and James Holmes

本章中英術語摘要：

class⇔類別；component⇔組件；concurrent⇔並行；exception⇔異常；garbage collection⇔垃圾回收；hierarchy⇔階層體系/繼承體系；host⇔主機；interface⇔(編程)介面/(GUI)界面；method (in Java)⇔函式；object⇔物件；overhead⇔額外開銷；parallel⇔平行；polymorphism⇔多型；process⇔行程；programming⇔編程/程式設計；synchronization⇔同步；thread⇔緒/緒程；type⇔型別；wrapper⇔包覆器；

人類的悠悠歷史，正反映到簡短的編程史話當中。早期人類社會從矇昧階段突然跳脫出來，編程技術也是如此。偉大文明興衰起落，編程語言也是如此。人類踏過國家的興起和衰敗，前進並發展。同樣地，不斷有新語言取代舊語言，使得編程技術的精煉和改進持續進行。

綜觀人類歷史，處處有重大事件。羅馬帝國衰微、1066 年大不列顛被入侵、第一次核爆，都改變了整個世界。編程語言也有同樣的事實，儘管規模較小。例如 FORTRAN 的發明永遠改變了計算機被程式驅使（programmed）的方式。另一個這般重大的事件就是 Java 的誕生。

Java 是編程工作進入 Internet 紀元的重要里程碑。它被刻意設計成：產生的程式可在任何有網路連線的地方執行。Java「只寫一次，到處執行」（write once, run anywhere）的哲學定義了新一代編程範型（programming paradigm）。Gosling 等人最初視之為處理小量問題的解決方案，如今儼然形成為下一代程式員定義編程風貌的一股力量。Java 如此根本性地改變了吾人對編程工作的思維，使得電腦語言的歷史被劃分為兩個紀元：「Java 前」和「Java 後」。

「Java 前」的世界裡，程式員產生的程式只能在獨立機器平台上執行。「Java 後」的世界裡，程式員能夠針對高度分散及網路化的環境來開發程式。程式員考量的不再是單一電腦，取而代之的是網路即電腦。如今我們考量的是伺服器（servers）、客戶端（clients）、主機（hosts）。

儘管 Java 的發展是因 Internet 的需求而推動，但它並不僅僅是一種「網路語言」。相反地，它是針對現代化網路世界而設計的一種全方位、多用途的程式語言。這表示 Java 適用於所有類型的編程工作。雖然有時候它被其強大的網路能力奪去光彩，但 Java 其實結合了許多足以提升編程藝術的創新功能，它們持續影響著整個業界。例如 C# 的某些觀點就是以 Java 開創的基礎為根據。

整本書裡頭，我們透過「把 Java 套用於應用程式各層面」的方式來展現 Java 的廣泛潛能。有些應用程式展現出 Java 在網路特性之外的強大威力，我們稱其為「純淨碼」（pure code）。之所以如此稱呼，因為它們展現了 Java 語法和設計哲學的豐富性。其他範例則是用來說明，使用 Java 語言和其 API classes 來開發複雜精巧的網路程式是多麼容易。整體而言，這些應用程式展現了 Java 的強大威力和寬廣視野。

開始探索 Java 之前，我們將在第一章花點時間指出，使 Java 成為卓越編程語言的一些特性。這些特性反映出我們所謂的「Java 風華」。

1.1. 簡單型別與物件：適當的平衡

Simple Types and Objects: The Right Balance

物件導向編程語言（OOPL）設計者所面臨的最大挑戰之一，就是如何處理 object（物件）

和 simple type（簡單型別）之間的兩難局面。這正是問題所在。從純理論觀點出發，每個 data type 都該是個 object，而每個 type 都應該源自共同的 parent object。這就可以使所有 data types 共享相同的繼承特性，以相同方式運作。問題是一旦讓 **int** 或 **double** 之類的 simple types 成為 object，可能會因為 object 機制帶來的額外開銷而減低效率。由於 simple types 常被拿來控制迴圈和條件述句，額外開銷會造成廣泛的負面效果。訣竅是在「所有東西都是 object」的渴望和「效率錙銖必較」的現實之間取得適當平衡。

Java 以一個優雅方式解決了 object 和 simple type 之間的問題。首先它定義八個 simple types：**byte**、**short**、**int**、**long**、**char**、**float**、**double** 和 **boolean**。這些 types 都會直接轉譯為二進制數值（binary values）。這麼一來 **int** 變數就可以讓 CPU 直接運算，不增加任何負擔。Java 的 simple types 就像其他語言那樣快速又有效率。因此，一個由 **int** 變數控管的 **for** 迴圈就能夠以最快速度執行，不會受到任何「object 相關議題」的限制。

除了 simple types，Java 的其他所有 types 都是從唯一的 superclass **Object** 衍生出來的 objects。所以其他所有 types 都可以共享那些繼承而來的機能。例如所有 objects 都擁有 **toString()**，因為那是 **Object** 定義的一個函式（method）。

由於 simple types 不是 objects，Java 得以自由地以不同方式處理 objects 和 nonobjects。這正是 Java 設計風華之展現。Java 中的所有 objects 都透過某個 reference 被存取，不像 simple types 那樣被直接存取。因此您的程式絕不會直接操作某個 object。透過這種方式，就能夠帶來像「垃圾回收」（garbage collection）這樣的好處。由於所有 objects 都經由 reference 進行存取，垃圾回收機制得以被有效實現出來：一旦某個 object 沒有被參考（引用），它就會被回收。另一個好處是，一個指向 type **Object** 的 object reference，可以指向系統中的任何一個 object。

當然，透過 reference 存取 object，會增加一些額外開銷，因為 reference 本質上是個位址（亦即指標），每個 object 都透過位址間接存取。儘管最新的 CPU 可以高效率處理間接存取，然而「間接存取」永遠比不上在 simple types 情況中對資料「直接處理」來得快。

儘管 simple types 相當有效率，有時候我們還是需要它的 object 等價物。例如您可能想要在執行期間產生一系列整數，而且希望不再需要它們時可以將之回收。為了處理這類狀況，Java 定義出像 **Integer** 和 **Double** 之類的 simple type wrapper（簡單型別包覆器）。這些包覆器讓 simple type 必要時有能力參與 objects 階層體系的運作。

面對 object 和 simple type，Java 的解決方式是抓出適當平衡點。它允許您開發高效率程式，也允許您使用 object model（物件模型）— 只要您不介意它對 simple types 的效率帶來負面影響。

1.2. 透過垃圾回收機制管理記憶體

Memory Management Through Garbage Collection

以垃圾回收機制（garbage collection）做為記憶體管理技術，早已行之有年，但 Java 賦予了它新生命。在 C++ 語言中，記憶體必須手動管理，程式員應該明確釋放不再使用的 objects。這正是問題的來源之一，因為我們常常忘記釋放一個不再需要的資源，或不小心釋放了還需用到的資源。

Java 避免這種問題的作法是：由它為您管理記憶體。這項作業將以高效率方式完成，因為 Java 的所有 objects 都透過 reference 被存取，一旦垃圾回收機制發現某個 object 未被引用（參照），就知道它已經不會再被使用，也就可以將它回收。如果 Java 允許 objects 被直接操作（像 simple types 那樣），如此高效率的垃圾回收機制可能就無法實現了。

Java 對垃圾回收機制的運用，反映出 Java 普遍存在的一種哲學。Java 設計者費盡心思要創造出一種語言，可避免其他語言發生的某些典型問題。透過垃圾回收機制，程式員不再會忘記釋放某個資源，或錯誤釋放某個使用中的資源。是的，垃圾回收機制攔截了整群問題。

1.3. 一個極佳的簡單化多緒模型

A Wonderfully Simple Multithreading Model

Java 設計者很早就瞭解到，未來編程一定會涉及「多緒多工」（multithreaded multitasking），因此應該在語言層次支援它。「多工」有兩種基本類型：process-based（以行程為基礎）和 thread-based（以緒程為基礎）。前者的最小排程單位是 process—基本上就是某個執行中的程式。因此 process-based 多工指的就是「一部電腦可在同一時間執行兩個或更多程式」。Thread-based 多工的最小排程單位是 thread—它定義程式中某個執行路線（execution path）。一個 process 可能包含多個 threads，而所謂「多緒程式」可以同時有多個部分被執行起來。

儘管 process-based 多工是大多數作業系統具備的一項能力，如果從語言層級支援 thread-based 多工，更是好處多多。舉個例子，C++ 沒有內建多緒編程能力，必須完全倚賴作業系統來處理多緒，這意味若要建立、啟動、同步化、結束某些緒程，需要多次呼叫作業系統。這使得 C++ 多緒程式碼不具可攜性。這也使得多緒在 C++ 程式中顯得不夠靈活。

由於 Java 內建多緒支援，其他語言中必須手動完成的許多工作在 Java 裡頭都被自動處理了。Java 多緒模型中最優雅的部分就是「同步化」（synchronization），那是根據兩項新創特性發展出來的。第一，所有 Java objects 都有內建監視器（build-in monitors），作用像是 mutually exclusive locks（互斥鎖）。特定時間內只有一個緒程可以擁有監視器。只要以關鍵字 **synchronized** 變更某個 method，就可以開啟上鎖（locking）特性。於是，一旦同步函式

（synchronized method）被呼叫，object 就被上鎖，其他想存取該 object 的緒程就必須等待。

Java 對「同步化」的第二個支援可以在 **Object** —所有其他 classes 的共同 superclass — 中找到。**Object** 宣告了以下的 synchronization methods：**wait()**、**notify()**和 **notifyAll()**。它們支援緒程間的溝通。因此所有 objects 也就都擁有了緒程溝通能力。一旦拿它們搭配某個 synchronized method，這些 methods 就允許（或說提供）某種高階的緒程互動方式。

由於多緒成為一項容易使用且內建於語言的能力，Java 改變了我們對程式基礎結構的認識。在 Java 之前，大多數程式員把程式想像成「擁有單一執行路線」的龐大結構。在 Java 之後，我們可以把程式視為許多「彼此可互動之平行工作（parallel tasks）」的集合。這種平行概念的變遷已經在電腦界造成廣泛影響，不過其最大衝擊或許是促進了軟體組件（software components）的運用。

1.4. 徹底結合異常處理機制

Fully Integrated Exceptions

異常（exception）的概念性框架早在 Java 之前就有了。的確有些語言結合了異常處理，例如 C++ 早在 Java 誕生數年前就做了這事。Java 的異常處理機制之所以重要，因為它是原始設計的一部份，而非附加品。是的，異常處理機制被徹底整合到 Java 內，構成了基礎特性之一。

Java 異常機制的一個關鍵觀點是：它的運用不屬於可選範圍。在 Java 中，透過異常機制來處理錯誤是一種慣例。這和 C++ 不同，C++ 並沒有將異常徹底結合到編程環境。試考慮開啟或讀取檔案時的常見情況，在 Java 中當這類操作發生任何錯誤，就有異常被丟出，而 C++ 開啟或讀取檔案的函式只會傳回特定錯誤代碼來回報狀況。這是因為 C++ 並非從最底層支援異常，所以其程式庫仍然倚賴錯誤代碼而非異常，您的程式因此必須經常手動檢查可能發生的錯誤。在 Java 程式中您只要簡單地以 **try/catch** 區塊包覆上述程式碼，任何錯誤便會被自動捕捉。

1.5. 有效支援多型

Streamlined Support for Polymorphism

多型（Polymorphism）是物件導向編程屬性，允許一個 interface 被多個 method 共用。Java 以多種特徵支援多型，其中兩種較受矚目。其一是每個 method（除非被標示為 **final**）都可被 subclass 覆寫（overridden）。其二是使用關鍵字 **interface**。請聽我進一步說明。

由於 superclass 內的 methods 可被其 derived classes 覆寫，因此可輕易建立 classes 階層體系

使其中 subclasses 全都是 superclass 的特化。記住，一個 superclass reference 可被拿來指向該 superclass 的任何 subclasses。此外，透過 superclass reference 呼叫 subclass object method，會自動執行起那個覆寫版本。因此 superclass 可用來定義某個 object 的形式並提供一份預設實作品，這份實作品可由 subclass 修改（客製化）以求最佳吻合其特定狀況。於是，同一個 interface（在此為 superclass 定義的那個）可被當作多重實作的根基。

當然啦，Java 把「一個 interface，多個 methods」的概念帶得更遠。它定義出關鍵字 **interface**，允許您將 class method 的宣告和實作完全區隔開來。儘管 interface 是抽象的，您還是可以宣告一個 reference 指向某 interface type，它可以被拿來指向任何實作出該 interface 之物。這是非常有威力的概念，因為它簡化並促進多型的運用。是的，只要某個 class 實作某個 interface，該 class object 就可被用在任何「需要該 interface 提供之機能」的程式碼上頭。舉個例子，假設有個 **MyIF** interface，想想以下這個 method：

```
void myMeth(MyIF ob) {
  // ...
}
```

任何實作出 **MyIF** interface 的 object 都可被傳遞給 **myMeth()**，不論該 object 具備其他哪些能力（如果有的話）。只要實作了 **MyIF**，**myMeth()**就可以操作它。

1.6. bytecode 帶來可攜性和安全性

Portability and Security Through Bytecode

儘管有了這麼多強大功能，如果少了 bytecode 這一重要但幾乎透明的部分，Java 也許不會在編程歷史上佔有如此重要的地位。就像所有 Java 程式員知道的，Java 編譯器輸出的並不是 CPU 可直接執行的機器碼，而是一組高度優化的可攜式指令，稱為 bytecode，由 JVM（Java 虛擬機器）執行。最初 JVM 只是個簡單的 bytecode 直譯器（interpreter），如今它已經可以編譯 bytecode 使其成為可執行碼。不論使用哪一種流程來執行 bytecode，其好處對於 Java 的成功都極為重要。

第一項好處是可攜性（portability）。Java 程式編譯成為 bytecode，就可以在任何電腦執行，不論那部電腦使用何種 CPU，只要該環境可執行 JVM 就可以達到目的。換句話說一旦某個特定環境實作出 JVM，就可以讓任何 Java 程式在那個環境下執行。開發者不需要針對不同的環境產生不同的執行碼。同一份 bytecode 就能夠在所有環境下運作。因此，透過 bytecode，Java 讓程式員有能力「只寫一次，到處執行」（write once, run anywhere）。

bytecode 具備的第二項好處是安全性（security）。由於 bytecode 執行作業全由 JVM 控管，

因此透過 JVM 就可以避免 Java 程式執行惡意操作，波及主機系統。這項「確保所在電腦安全性」的能力對 Java 的成就有決定性影響，因為有了這樣的能力才能造就 applet，那是一種精簡、可透過 Internet 被動態下載的程式，因此必須有「避免 applet 傷害系統」的機制。Bytecode 和 JVM 的結合提供了一個機制，使 applet 可被安全下載。坦白說，要是沒有 bytecode 的出現，今天的網路世界可能會是另一種完全不同的型態。

1.7. 豐富的 Java API

The Richness of the Java API

概念上，電腦語言由兩部分構成。第一部分是關鍵字和語法構成的語言特性，第二部分是給程式員使用，由各種 classes、interfaces、methods 構成的標準程式庫。儘管目前所有主流語言都提供龐大的程式庫資源，Java 規範的這一套卻因為其豐富性和多樣性而格外引人注目。Java 剛開發出來時，其程式庫只包含一套諸如 **java.lang**、**java.io** 和 **java.net** 的核心套件。隨著新版 Java 釋出，不斷加入新的 classes 和 packages。如今 Java 提供給程式員的功能已有相當驚人的數量。

打從一開始，Java 程式庫和其他語言程式庫的最大不同就在於，它對網路的支援。在 Java 誕生時期，其他語言如 C++ 者，並沒有（甚至現在仍然沒有）提供具備網路能力的標準程式庫。由於 Java 提供了可輕易處理連線工作和 Internet 運用的 classes，因此也對 Internet 的革命性發展帶來推波助瀾之效。有了 Java，Internet 也因此為所有程式員開放，而不僅止於需要網路的那些人。**java.net** 具備的功能改變了整個資訊產業。

Java 核心程式庫中另一個關鍵性的 package 就是 **java.awt**。它提供「抽象視窗工具箱」（Abstract Window Toolkit, AWT）。AWT 讓程式員得以產生可攜式 GUI 程式。也就是說運用 AWT classes 就可開發出視窗程式，其中有捲軸、核取方塊、選取鈕之類的標準 GUI 元素。有了 AWT，就可以開發出 GUI 程式並讓它在任何支援 Java 虛擬機器的環境中執行。Java 出現之前，這種 GUI 可攜性可說是前所未見。

Java 引入 AWT 這樣的東西，徹底改變了程式員對應用程式環境的思考邏輯。在 Java 之前，GUI-based 程式必須針對執行環境特別開發。例如一個 Windows 程式必須經過大幅修改才可以在 Apple 電腦上執行。有了 Java，程式員只需編寫一套程式即可，它在兩個環境下都能執行。由於定義了一個可攜式 GUI，Java 統一了編程環境。

近些年 Java 又在 AWT 之外加入一套輕量級套件：Swing，包含於 **javax.swing** 套件及其附屬套件。Swing 提供豐富的 GUI 控件，而且更加強了可攜性。如同本書許多範例所展現的那樣，

AWT 和 Swing 都讓程式員有能力創造出 GUI-based、具高度可用性和可攜性的應用程式。

如今 Java 程式庫已經從最初核心有了大幅成長。Java 的每一個新版本都會伴隨一些程式庫支援。不但新套件被加入，也有些新功能被加入既有套件中。Java 程式庫不斷演變，因為它很容易受到快速變遷的資訊環境所影響。這種適應性和變化能力也是 Java 風華的一部份。

1.8. Applet

就算以今天的標準來說，applet 也是 Java 最具革命性的性質之一，因為它允許生成可攜式、可被動態下載，並在瀏覽器上安全執行的程式。在 Java 出現之前，可執行內容一直都是備受質疑的主題，因為惡意程式可能會對客端電腦造成損害。再說，針對某 CPU 和 O.S. 編譯出來的程式碼，可能無法在另一系統中運作。Internet 上的 CPU 和 O.S. 種類繁多，產生一份唯一版本給所有環境去執行，顯然不切實際。Java applet 為以上兩個問題同時提供了解答。有了 applet，Web 程式員就能夠輕易為網頁加入動態內容，而不只是靜態的 HTML。Java applet 讓 Web 技術往前邁一大步，而且沒有回頭路。

除了改變我們對網頁內容的思考外，applet 還有另一重要作用 — 也許是副作用。它協助推動了組件軟體（component software）。Applet 是小程式，往往被拿來表現小巧的功能單元，而這正是組件做的事。一旦我們開始思考 applet，一小步就能夠觸及 Bean，然後超越它。今天，組件導向軟體架構（亦即應用程式由一群互動組件構成）已經大幅取代了過去那些典型的龐大模型。

1.9. 永不停息的革命

The Continuing Revolution

還有一個 Java 觀點也能夠反映出 Java 本身的風華，不過它並不真正是語言的一部分。是的，Java 帶來了一種樂於接受新點子的創新文化，以及一種讓這些新點子被快速消化吸收的過程。其他許多電腦語言的改變十分緩慢，Java 則是不斷地進化並調整，而且這個程序是開放的，透過 Java Community Process（JCP）開放給整個 Java 社群。JCP 提供了一種機制，使 Java 用戶能夠影響語言、工具及相關技術的未來方向。因此，真正使用這個語言的人，會投入到這個不斷發展的環境中。

從一開始，Java 就已經為編程技術做出突破性的重大變革，這項革命至今還未停息。Java 依然站在電腦語言發展的最前線。它是在電腦史上佔有一席之地的編程語言。

2

2.遞迴漸降算式解析器

A Recursive-Descent Expression Parser

By Herb Schildt

本章中英術語摘要：

applet⇔小程式；argument⇔引數；evaluate⇔評估/核算；expression⇔算式/表達式；exception⇔異常；loop⇔迴圈；parameter⇔參數；parse⇔解析；parser⇔解析器；recursive⇔遞迴；recursive descent⇔遞迴漸降；

如何寫出程式，接受一個輸入字串，內容類似(10-5)*3 這樣的數值算式，並估算出答案呢？如果程式員之間還有所謂的「能人」，想必只有那些人才知道該怎麼辦吧。多數程式員都對「高階語言如何將代數算式轉換為電腦可執行指令」的手法感到驚奇。這個程序就叫做「算式解析」（expression parsing），它是所有語言編譯器、直譯器、試算表，以及其他需要「把數值算式轉換為電腦可使用之某種形式」的基礎。

儘管不識此道者對此可能感到神祕不可思議，但「算式解析」是一個定義明確的工作，並且已有優雅的解答。由於問題有明確的定義，所以「算式解析」可根據嚴謹的代數規則來運作。本章將發展出所謂的「遞迴漸降解析器」（recursive-descent parser），以及評估數值算式時一切必要的支援函式。一旦掌握了 parser 運作方式，就可依照自己的需求進行加強和修改。

把 parser（解析器）當做本書第一個範例的原因，不僅在於其程式碼本身有用，也因為它可以闡明 Java 語言的能力與範疇。parser 算是一種「純淨碼」次系統，我意思是它既非網路導向，也不倚賴 GUI 介面，也不是 applet 或 servlet。這是一種您可能預期在 C 或 C++ 但非 Java 中看到的程式類型。由於 Java 是一種革命性力量，徹底改變了我們對 Internet 的編程方式，我們有時候可能會忘記它其實並不被限制於那樣的環境。也就是說 Java 是一種可應用於任何編程型態的全方位語言。本章開發的 parser 將證明這一點。

2.1. 算式/表達式

Expressions

parser 處理的對象是個算式（表達式），所以它首先必須瞭解，算式由什麼東西組成。儘管算式有許多不同類型，本章只處理其中一種類型：數值算式（numeric expressions）。我們處理的數值算式由以下各項組成：

- 數字
- 運算子 +,-,/,*,^,%,=
- 括號
- 變數

此處的 ^ 代表指數運算（不是 Java 的 XOR 運算），= 是賦值運算子（assignment operator）。這些項目都可根據代數規則組合到算式中。下面就是一些例子：

```
10 - 8
(100 - 5) * 14/6
a + b - c
10^5
a = 10 - b
```

假設上述每一個運算子的優先序（precedence）如下。優先序相等者，由左至右運算。

最高	+,-（一元運算）
	^
	*,/,%
	+,-（二元運算）
最低	=

本章開發出來的 parser 有數個限制。第一，所有變數名稱都只是單一字母（也就是 A~Z 共 26 個變數名稱可用），無大小寫之分（a 和 A 會被視為同一變數）。第二，所有數值都被當作 **double** 運算，不過您可以輕易修改 parser，讓它處理其他類型的數值。最後，為保持邏輯上的簡潔，以及為了容易理解，程式只對基本錯誤進行檢驗。

2.2. 解析算式：問題所在

Parsing Expressions: The Problem

如果您不曾對「算式解析」多加思索，可能會以為那是件簡單的工作。事實並非如此。您可以試著評估以下這個例子：

```
10 - 2 * 3
```

您知道這個算式的結果為 4。儘管可以輕易寫出一個程式計算特定算式的結果，但如何開發出一個程式，面對任意算式仍能求得正確的答案呢？一開始您可能會想出這樣的演算法：

```
a = get first operand          //取得第一個運算元
while(operands present) {      //當運算元存在
  op = get operator            //取得運算子
  b = get second operand       //取得第二個運算元
  a = a op b                   //執行運算
}
```

這個解法取得第一運算元、運算子，和第二運算元，然後執行第一次運算，再取得下一個運算子和運算元，執行下一個運算，依此類推。如果您嘗試使用這樣的基本手法，算式 10-2*3 會得到 24（也就是 8*3 的結果），而不是正確答案 4，因為這個程序沒有處理運算子優先序。是的，您不能只是由左而右地取運算元和運算子，因為根據代數規則，乘法必須先於減法被執行。有些初學者認為這問題很容易克服，而有時候在某些受限狀況下的確如此。不過當您加上括號、指數運算、變數、一元運算子，問題只會更加複雜。

儘管有不少方式可寫出能夠處理算式的程式碼，但這裡所發展的，是憑一人之力就可以輕鬆進行的方式，我們稱之為「遞迴漸降式」（recursive-descent）parser，您將在本章瞭解到為

何以這個名字稱呼它。其他的 parser 有些用上了複雜的資料表，該表通常由另一個電腦程式產生，是謂「表格驅動式（table-driven）parsers」。

2.3. 解析一個算式

Parsing an Expression

算式的解析（parse）及核算（evaluate）方法不少。遞迴漸降式 parser 採取的辦法，是把算式當作一組遞迴資料結構（recursive data structure）來思考。也就是說，算式由其本身定義而成。假設算式只能使用 +,-,*,/ 和括號，那麼所有算式可根據以下規則定義出來：

```
expression → term [+term ] [ - term ]          //算式→項 [+項] [-項]
term → factor [*factor ] [/factor ]            //項→因子 [*因子] [/因子]
factor → variable,number,or (expression)       //因子→變數，數值，或（算式）
```

中括號代表可有可無，'→' 表示「生成」。事實上這些規則常被稱為算式生成規則（production rules）。因此「項」的定義您可以解釋為「項會生成 "因子乘以因子" 或 "因子除以因子"」。請注意，運算子優先序隱含在算式的定義中。

這裡有個例子。算式：

```
10 + 5 * B
```

有兩項：10 和 5*B。其中第二項包含了兩個因子：5 和 B，一個是數值，一個是變數。另一個算式：

```
14 * (7 - C)
```

擁有兩個因子：14 和 (7-C)，一個是數值，另一個是被括號括住的算式。後者又包含兩項：一個數值和一個變數。

這樣的程序形成了遞迴漸降式 parser 的基礎，那是一組相互遞迴的函式集（recursive methods set），以連鎖形式（chainlike fashion）運作，實作出上述生成規則。在每一個適當步驟中，parser 會依代數上的正確次序執行特定運算。為了瞭解生成規則如何解析算式，讓我們演練底下這個式子：

```
9/3 - (100 + 56)
```

您會採取以下一連串處理：

1. 取得第一項，9/3。
2. 取得每個因子，並進行整數除法。結果為 3。

3. 取得第二項，(100 + 56)。此時開始以遞迴方式分析這個子算式。

4. 取得每一項，進行加法運算。結果為 156。

5. 傳回第二項的遞迴運算結果。

6. 拿 3 減去 156，得到最後答案為 -153。

如果此時您生出一些疑惑，別在意，這是個需要花點功夫才能習慣的複雜概念。關於算式的遞迴觀點，必須記住兩個基本重點。第一，運算子的優先序隱含在事先定義的生成規則中。第二，這個解析並核算（evaluate）算式的辦法，和人類核算數學算式的辦法十分類似。

本章稍後發展出兩個 parsers。第一個會根據字面內容（都是浮點數）以 **double** type 進行解析及核算。這個 parser 用來說明解析過程中的遞迴漸降原理。第二個 parser 則添加變數處理能力。

2.4. 深入剖析算式結構

Dissecting an Expression

為了核算某一算式，parser 必須被餵以個別成分。例如以下算式：

```
A * B - (W + 10)
```

就包含了一些個別成分：`A,*,B,-,(,W,+,10,)`。在專業用語中，算式的每一個組成元素稱為 token（語彙單元），每一個 token 代表算式之中不可分割的最小單位。由於切割 token 是解析工作的重要基礎，讓我們先仔細看看它。

為了將算式分割為 token，您需要一個函式，可以從頭到尾不斷將算式的每一個 token 傳回。這個函式還必須能夠判別 token 類型，並察覺結束狀況。此處我們稱它為 **getToken()**。

本章的兩個 parsers 都被封裝在 **Parser** class 中。稍後才有這個 class 的細部描述，但現在必須先將其開頭部分呈現出來，以便解釋 **getToken()** 的運作。**Parser** class 一開始先定義一些 **final** 變數和資料欄：

```
#001 class Parser {
#002   // 這些是 token 類型
#003   final int NONE = 0;
#004   final int DELIMITER = 1;
#005   final int VARIABLE = 2;
#006   final int NUMBER = 3;
#007
#008   // 這些是語法錯誤類型
#009   final int SYNTAX = 0;
#010   final int UNBALPARENS = 1;
```

```
#011   final int NOEXP = 2;
#012   final int DIVBYZERO = 3;
#013
#014   // 這個 token 代表算式結尾（end-of-expression）
#015   final String EOE = "\0";
#016
#017   private String exp;     // 指向算式字串
#018   private int expIdx;     // 算式的目前索引位置
#019   private String token;   // 用以儲存目前 token
#020   private int tokType;    // 用以儲存 token 類型
```

Paser 首先定義出代表 token 類型的數值。解析一個算式時，每個 token 都必須隸屬某種類型。本章開發的 parser 只用到三種 token 類型：變數、數值，定義符號（delimiter），分別以 **VARIABLE**、**NUMBER** 和 **DELIMITER** 表示。**DELIMITER** 類型同時用於運算子和括號。**NONE** 類型保留給未經定義的 token 使用。

接下來，**Paser** 定義一些數值用來代表解析和核算過程中可能發生的各種錯誤。**SYNTAX** 代表畸形算式。**UNBALPARENS** 代表括號不對稱。**NOEXP** 代表 parser 執行時沒有出現任何需要解析的算式。**DIVBYZERO** 表示「除以零」。

最後那個 **EOE**，用來表示已到達算式尾端。

exp 是個字串（嚴格說是個 reference to string），其中儲存「待解析算式」。因此 **exp** 會指向諸如 "10+4" 這樣的字串。該字串中的「下一個 token」索引值（一開始當然是 0）將被保存在 **expIdx** 裡頭。取得的 token 被儲存於 **token** 變數，token 類型則被儲存於 **tokType**。這些資料欄都是 **private**，因為它們只供 parser 使用，不該被外界程式碼變更。

以下就是 **getToken()**。每被呼叫一次它就由 **exp** 所指字串中的 **expIdx** 所指位置上取得算式下一個 token。換句話說每當 **getToken()** 被呼叫，它就會從 **exp[expIdx]** 取出下一個 token。取得的 token 放入 **token** 變數，token 類型放入 **tokType**。以下將 **getToken()** 用到的 **isDelim()** 也一併呈現出來。

```
#001 // 獲取下一個 token
#002 private void getToken()
#003 {
#004   tokType = NONE;
#005   token = "";
#006
#007   // 檢查是否為算式尾端
#008   if(expIdx == exp.length()) {
#009     token = EOE;
#010     return;
```

```
#011   }
#012
#013   // 跳過空白字元（white space）
#014   while(expIdx < exp.length() &&
#015     Character.isWhitespace(exp.charAt(expIdx))) ++expIdx;
#016
#017   // 檢查是否為算式結尾空白字元
#018   if(expIdx == exp.length()) {
#019     token = EOE;
#020     return;
#021   }
#022
#023   if(isDelim(exp.charAt(expIdx))) {          // 是個運算子
#024     token += exp.charAt(expIdx);
#025     expIdx++;
#026     tokType = DELIMITER;
#027   }
#028   else if(Character.isLetter(exp.charAt(expIdx))) {  // 是個變數
#029     while(!isDelim(exp.charAt(expIdx))) {
#030       token += exp.charAt(expIdx);
#031       expIdx++;
#032       if(expIdx >= exp.length()) break;
#033     }
#034     tokType = VARIABLE;
#035   }
#036   else if(Character.isDigit(exp.charAt(expIdx))) {    // 是個數值
#037     while(!isDelim(exp.charAt(expIdx))) {
#038       token += exp.charAt(expIdx);
#039       expIdx++;
#040       if(expIdx >= exp.length()) break;
#041     }
#042     tokType = NUMBER;
#043   }
#044   else {    // 若是未知字元，就結束算式
#045     token = EOE;
#046     return;
#047   }
#048 }
#049
#050 // 如果 c 是個定義符號（delimiter），就傳回 true
#051 private boolean isDelim(char c)
#052 {
#053   if((" +-/*%^=()".indexOf(c) != -1))
#054     return true;
#055   return false;
#056 }
```

讓我們進一步觀察 **getToken()**。一開始的初始化作業後，**getToken()** 看看 **expIdx** 是否和

exp.length() 相等，以檢驗目前是否已經處理至算式尾端。由於 **expIdx** 是個索引，如果它和待分析算式之字串長度相等，就代表整個算式已被完全解析過了。

如果該算式還可取得更多 tokens，**getToken()** 就會首先跳過前頭任何空白字元。如果這樣下去的最後一個空白字元是算式尾端，就傳回 **EOE**。否則一旦跳過前端空白，**exp[expIdx]** 將內含某個數字，變數，或運算子。如果下個字元是運算子，就讓 **token** 儲存一個字串並將 **DELIMITER** 存入 **tokType**。如果下個字元是英文字母，會被當作變數，於是便讓 **token** 儲存一個字串並將 **tokType** 指定為 **VARIABLE**。如果下個字元是數字，就讀取整個數值並將它以字串形式存入 **token**，類型則指定為 **NUMBER**。最後，如果下個字元不在辨識範圍內，就把 **token** 指派為 **EOE**。

為了讓 **getToken()** 看起來較簡潔，這裡省略了一些錯誤檢驗，並做了一些假設。例如，任何未被認可的字元只要其前面是個空白字元，就會結束算式。又如，在這個版本中變數名稱無長度限制，但只有第一個字母有意義。如果您的特殊應用有所需求，您可以加上更多錯誤檢查和其他細節。

為了更進一步瞭解 token 分割程序，試分析下面這個算式會傳回什麼樣的 token 和 token 類型：

```
A + 100 - (B * C) /2
```

token	token 類型
A	VARIABLE
+	DELIMITER
100	NUMBER
-	DELIMITER
(	DELIMITER
B	VARIABLE
*	DELIMITER
C	VARIABLE
)	DELIMITER
/	DELIMITER
2	NUMBER

請記住，**token** 存放的永遠都是字串，即使它只內含單一字元。

最後一點：儘管 Java 有諸如 **StringTokenizer** 這樣的 class 提供了一些有用的 token 切割功能，但對 parser 而言，如果能夠完全以 **getToken()** 這樣的專屬切割器自理，最好不過了。

2.5. 一個簡易型算式解析器

A Simple Expression Parser

這就是第一版 parser。它只能核算由字面常數（literal）、運算子和括號構成的算式。雖然 **getToken()** 可以處理變數，但 parser 對它什麼也不做。一旦您瞭解這個簡化版本如何運作之後，我們將為它加上變數處理能力。

```
#001 /*
#002    這個模組內含遞迴漸降式 parser，但不處理變數
#003 */
#004
#005 // 為 parser 錯誤狀況所準備的異常類別（exception class）
#006 class ParserException extends Exception {
#007   String errStr; // 用來描述錯誤
#008
#009   public ParserException(String str) {
#010     errStr = str;
#011   }
#012
#013   public String toString() {
#014     return errStr;
#015   }
#016 }
#017
#018 class Parser {
#019   // 這些是 token 類型
#020   final int NONE = 0;
#021   final int DELIMITER = 1;
#022   final int VARIABLE = 2;
#023   final int NUMBER = 3;
#024
#025   // 這些是語法錯誤類型
#026   final int SYNTAX = 0;
#027   final int UNBALPARENS = 1;
#028   final int NOEXP = 2;
#029   final int DIVBYZERO = 3;
#030
#031   // 這個 token 代表「算式尾端」（end-of-expression）
#032   final String EOE = "\0";
#033
#034   private String exp;     // 指向算式字串
#035   private int expIdx;     // 算式的目前索引位置
#036   private String token;   // 用以儲存目前的 token
#037   private int tokType;    // 用以儲存 token 類型
#038
```

```
#039  // parser 進入點
#040  public double evaluate(String expstr) throws ParserException
#041  {
#042    double result;
#043    exp = expstr;
#044    expIdx = 0;
#045
#046    getToken();
#047    if(token.equals(EOE))
#048      handleErr(NOEXP); // 未見任何算式
#049
#050    // 解析與核算整個算式
#051    result = evalExp2();
#052
#053    if(!token.equals(EOE)) // 最後一個 token 一定是 EOE
#054      handleErr(SYNTAX);
#055
#056    return result;
#057  }
#058
#059  // 加或減兩個「項」(terms)
#060  private double evalExp2() throws ParserException
#061  {
#062    char op;
#063    double result;
#064    double partialResult;
#065
#066    result = evalExp3();
#067
#068    while((op = token.charAt(0)) == '+' || op == '-') {
#069      getToken();
#070      partialResult = evalExp3();
#071      switch(op) {
#072        case '-':
#073          result = result - partialResult;
#074          break;
#075        case '+':
#076          result = result + partialResult;
#077          break;
#078      }
#079    }
#080    return result;
#081  }
#082
#083  // 乘或除兩個「因子」(factor)
#084  private double evalExp3() throws ParserException
#085  {
#086    char op;
```

```
#087    double result;
#088    double partialResult;
#089
#090    result = evalExp4();
#091
#092    while((op = token.charAt(0)) == '*' ||
#093          op == '/' || op == '%') {
#094      getToken();
#095      partialResult = evalExp4();
#096      switch(op) {
#097        case '*':
#098          result = result * partialResult;
#099          break;
#100        case '/':
#101          if(partialResult == 0.0)
#102            handleErr(DIVBYZERO);
#103          result = result / partialResult;
#104          break;
#105        case '%':
#106          if(partialResult == 0.0)
#107            handleErr(DIVBYZERO);
#108          result = result % partialResult;
#109          break;
#110      }
#111    }
#112    return result;
#113  }
#114
#115  // 處理指數（exponent）
#116  private double evalExp4() throws ParserException
#117  {
#118    double result;
#119    double partialResult;
#120    double ex;
#121    int t;
#122
#123    result = evalExp5();
#124
#125    if(token.equals("^")) {
#126      getToken();
#127      partialResult = evalExp4();
#128      ex = result;
#129      if(partialResult == 0.0) {
#130        result = 1.0;
#131      } else
#132        for(t=(int)partialResult-1; t > 0; t--)
#133          result = result * ex;
#134    }
#135    return result;
```

```
#136   }
#137
#138   // 處理一元運算 + 或 -
#139   private double evalExp5() throws ParserException
#140   {
#141     double result;
#142     String  op;
#143
#144     op = "";
#145     if((tokType == DELIMITER) &&
#146        token.equals("+") || token.equals("-")) {
#147       op = token;
#148       getToken();
#149
#150     }
#151     result = evalExp6();
#152
#153     if(op.equals("-")) result = -result;
#154
#155     return result;
#156   }
#157
#158   // 處理「括號內的算式」
#159   private double evalExp6() throws ParserException
#160   {
#161     double result;
#162
#163     if(token.equals("(")) {
#164       getToken();
#165       result = evalExp2();
#166       if(!token.equals(")"))
#167         handleErr(UNBALPARENS);
#168       getToken();
#169     }
#170     else result = atom();
#171
#172     return result;
#173   }
#174
#175   // 取得一個數值（number）的值（value）
#176   private double atom() throws ParserException
#177   {
#178     double result = 0.0;
#179
#180     switch(tokType) {
#181       case NUMBER:
#182         try {
#183           result = Double.parseDouble(token);
#184         } catch (NumberFormatException exc) {
```

```
#185          handleErr(SYNTAX);
#186        }
#187        getToken();
#188        break;
#189      default:
#190        handleErr(SYNTAX);
#191        break;
#192    }
#193    return result;
#194  }
#195
#196  // 處理錯誤狀況
#197  private void handleErr(int error) throws ParserException
#198  {
#199    String[] err = {
#200      "Syntax Error",
#201      "Unbalanced Parentheses",
#202      "No Expression Present",
#203      "Division by Zero"
#204    };
#205
#206    throw new ParserException(err[error]);
#207  }
#208
#209  // 取下一個 token
#210  private void getToken()
#211  {
#212    tokType = NONE;
#213    token = "";
#214
#215    // 檢查是否為算式尾端
#216    if(expIdx == exp.length()) {
#217      token = EOE;
#218      return;
#219    }
#220
#221    // 跳過空白字元（white space）
#222    while(expIdx < exp.length() &&
#223      Character.isWhitespace(exp.charAt(expIdx))) ++expIdx;
#224
#225    // 檢查是否為算式尾端的空白字元
#226    if(expIdx == exp.length()) {
#227      token = EOE;
#228      return;
#229    }
#230
#231    if(isDelim(exp.charAt(expIdx))) { // 是個運算子
#232      token += exp.charAt(expIdx);
```

```
#233      expIdx++;
#234      tokType = DELIMITER;
#235    }
#236    else if(Character.isLetter(exp.charAt(expIdx))) { // 是個變數
#237      while(!isDelim(exp.charAt(expIdx))) {
#238        token += exp.charAt(expIdx);
#239        expIdx++;
#240        if(expIdx >= exp.length()) break;
#241      }
#242      tokType = VARIABLE;
#243    }
#244    else if(Character.isDigit(exp.charAt(expIdx))) { // 是個數值
#245      while(!isDelim(exp.charAt(expIdx))) {
#246        token += exp.charAt(expIdx);
#247        expIdx++;
#248        if(expIdx >= exp.length()) break;
#249      }
#250      tokType = NUMBER;
#251    }
#252    else { // 遇到未知字元便結束整個算式
#253      token = EOE;
#254      return;
#255    }
#256  }
#257
#258  // 如果 c 是個定義符號（delimiter），就傳回 true
#259  private boolean isDelim(char c)
#260  {
#261    if((" +-/*%^=()".indexOf(c) != -1))
#262      return true;
#263    return false;
#264  }
#265
#266 }
```

請注意一開始宣告的 **PaserException** class。這是本程式所用的異常型別，如果算式處理過程中發生錯誤，parser 就丟出這種異常。這個異常必須由使用此一 parser 的程式去處理。

這個 parser 就像您所看到的那樣，可以處理運算子 `+,-,*,/,%`。此外它也可以處理整數指數運算（`^`）和一元負號運算。它還可以正確處理帶有括號的情況。

要使用這個 parser，首先得建立一個 **Parser** object，然後呼叫其 **evaluate()**，把您想要核算的算式字串當作引數（argument）傳進去。核算結果會被傳回。由於 **Parser** 會在錯誤發生時丟出一個 **ParserException** 異常，因此您的應用程式必須處理這樣的異常。以下程式示範 parser 的運用：

```
#001 // 示範 parser 的運用
#002 import java.io.*;
#003
#004 class PDemo {
#005   public static void main(String args[])
#006     throws IOException
#007   {
#008     String expr;
#009
#010     BufferedReader br = new
#011       BufferedReader(new InputStreamReader(System.in));
#012     Parser p = new Parser();
#013
#014     System.out.println("Enter an empty expression to stop.");
#015
#016     for(;;) {
#017       System.out.print("Enter expression: ");
#018       expr = br.readLine();
#019       if(expr.equals("")) break;
#020       try {
#021         System.out.println("Result: " + p.evaluate(expr));
#022         System.out.println();
#023       } catch (ParserException exc) {
#024         System.out.println(exc);
#025       }
#026     }
#027   }
#028 }
```

執行結果如下：

```
Enter an empty expression to stop.
Enter expression: 10-2*3
Result: 4.0

Enter expression: (10-2)*3
Result: 24.0

Enter expression: 10/3.5
Result: 2.857142857142857
```

2.5.1. 認識解析器（Understanding the Parser）

讓我們詳細觀察 **Parser**。待被核算之算式字串由 **exp** 指出來。每當 **evaluate()** 被呼叫時 **exp** 資料欄都應該設妥內容。請記住，這個 parser 意欲核算的算式是被置於 Java 標準字串中，例如以下這些字串就合乎要求：

```
"10 - 5"
"2 * 3.3 / (3.1416 * 3.3)"
```

指向 **exp** 的現行索引值存放在 **expIdx** 中。解析工作一開始 **expIdx** 會先被設定為 0，而後其值隨著 parser「走過」整個算式而增加。**token** 保存著目前的 token，**tokType** 則存放 token 的類型。

parser 的進入點就是 **evaluate()**。呼叫它時必須傳入一個內含待被分析之算式字串。**evalExp2()** 至 **evalExp6()** 這幾個函式，搭配 **atom()**，構成了我們所說的這個遞迴漸降式 parser。它們實現了先前討論的算式生成規則的一個強化集。每個函式最前面的註解說明了它們將執行的功能。下一版 parser 還會加上 **evalExp1()**。

handleErr()負責處理算式中的語法錯誤。**getToken()**和 **isDelim()**就像先前描述的那樣，把算式分解成各個構成元素。parser 使用 **getToken()**取得算式中的 tokens，並從算式起始處動手，直到算式結束為止。程式會根據取得 token 的類型而採取對應的作業。

為了確切瞭解這個 parser 如何核算算式，試以底下算式為例進行演練：

```
10 - 3 * 2
```

當 parser 的進入點 **evaluate()**被呼叫，它取得第一個 token。如果這個 token 是 **EOE**，就表示待被核算的是個空字串（null string），這會產生 **NOEXP** 錯誤。本例中的第一個 token 內含 `10`。接下來呼叫 **evalExp2()**，其中呼叫 **evalExp3()**，後者又呼叫 **evalExp4()**，後者再呼叫 **evalExp5()**。**evalExp5()**檢查 token 是否為一元正值或一元負值運算，如果不是（一如本例）就呼叫 **evalExp6()**。這時候 **evalExp6()**如果不是遞迴呼叫 **evalExp2()**（當出現「括號內的子

算式」時），就是呼叫 **atom()**取得真實數值。由於本例的 token 不是個左括號，因此會執行 **atom()**並獲得數值 10。接下來再取另一個 token，操作程序相同，回到鏈狀起點。此時的 token 是 -，於是整組 methods 回到 **evalExp2()**。

接下來發生的事情非常重要。由於目前的 token 是 -，因此被存入 **op**。parser 繼續取下一個 token，也就是 3，而鏈狀漸降過程再度展開。和先前一樣，最終進入 **atom()**。於是數值 3 被傳回給 **result**，並讀取下一個 token *。這會造成回至鏈狀中的 **evalExp3()**，其中會讀取最後一個 token 2。這時候第一個算術運算浮現：2 乘以 3。運算結果回傳至 **evalExp2()**，於是又執行減法，導出答案為 4。儘管第一次看到這樣的程序可能覺得有點複雜，不過繼續拿其他例子演練一下，就可以證明這個 parser 總是能夠正確完成任務。

如果解析過程中發生錯誤，就呼叫 **handleErr()**。它會丟出一個 **ParserException**，用來描述錯誤狀況。這個異常又會從 **evaluate()**丟出，讓 parser 使用者去處理。

這個 parser 很適合拿來當作一個簡易桌上計算器，類似稍早那個示例程式的所做所為。在它能夠被運用於某個電腦語言、資料庫，或更精緻的計算器之前，它還需要變數處理能力。這正是下一節的主題。

2.6. 為解析器添加變數

Adding Variables to the Parser

所有編程語言、多數計算器，以及試算表軟體，都會使用變數保存數據以供稍後運用。因此在 parser 被做為這類應用之前，必須先為它加入變數處理能力。您必須為 parser 加上一些東西才能達到這樣的目的。首先當然是變數本身。正如先前所言，我們將以字母 A~Z 作為變數名稱。這些變數被儲存在 **Parser** class 內的一個陣列中。每個變數都在「擁有 26 個 **double** 元素」的陣列中佔一個位置。因此，請為 **Parser** class 加上這些資料欄：

```
// 供變數所用的陣列
private double vars[] = new double[26];
```

陣列元素會在 **Parser** object 被具現化（instantiated）時自動獲得初值 0。

您還需要一個函式，用來查詢某變數的實際值。由於變數以 A~Z 命名，因此只要將變數名稱的 ASCII 值減去 A 的 ASCII 值，就可輕易獲得陣列 **vars** 的索引。下面的 **findVar()**完成了這項工作：

```
#001 // 傳回變數值
#002 private double findVar(String vname) throws ParserException
#003 {
```

```
#004   if(!Character.isLetter(vname.charAt(0))){
#005     handleErr(SYNTAX);
#006     return 0.0;
#007   }
#008   return vars[Character.toUpperCase(vname.charAt(0))-'A'];
#009 }
```

這個函式其實可以接受像 **A12** 或 **test** 這樣的長變數名稱，不過只有第一個字母有意義。您可以按自己的需要變更這項功能。

您也必須修改 **atom()**以便同時處理數值和變數。新版本如下：

```
#001 // 取得數字或變數的實際值
#002 private double atom() throws ParserException
#003 {
#004   double result = 0.0;
#005
#006   switch(tokType) {
#007     case NUMBER:
#008       try {
#009         result = Double.parseDouble(token);
#010       } catch (NumberFormatException exc) {
#011         handleErr(SYNTAX);
#012       }
#013       getToken();
#014       break;
#015     case VARIABLE:
#016       result = findVar(token);
#017       getToken();
#018       break;
#019     default:
#020       handleErr(SYNTAX);
#021       break;
#022   }
#023   return result;
#024 }
```

嚴格說來，附加部分就是「讓 parser 得以正確使用變數」所需的一切；然而，這裡沒什麼辦法可讓變數被賦予一個值。為了讓變數得以被賦值，parser 必須能夠處理賦值運算子 =。為了實現賦值行為，我們將在 **Parser** class 內加上另一個函式，名為 **evalExp1()**，由它來啟動遞迴漸降鏈。也就是說 **evaluate()**應該呼叫它（而不再是 **evalExp2()**）。其程式碼如下：

```
#001 // 處理賦值（assignment）命令
#002 private double evalExp1() throws ParserException
#003 {
#004   double result;
#005   int varIdx;
```

```
#006   int ttokType;
#007   String temptoken;
#008
#009   if(tokType == VARIABLE) {
#010     // 儲存原來的 token
#011     temptoken = new String(token);
#012     ttokType = tokType;
#013
#014     // 計算變數的索引值
#015     varIdx = Character.toUpperCase(token.charAt(0)) - 'A';
#016
#017     getToken();
#018     if(!token.equals("=")) {
#019       putBack(); // 傳回目前的 token
#020       // 取回原來的 token － 不是一個賦值命令
#021       token = new String(temptoken);
#022       tokType = ttokType;
#023     }
#024     else {
#025       getToken(); // 取得 exp 的下一成分
#026       result = evalExp2();
#027       vars[varIdx] = result;
#028       return result;
#029     }
#030   }
#031
#032   return evalExp2();
#033 }
```

evalExp1() 必須事先判定是否真要執行賦值作業。這是因為變數名稱總是出現在賦值運算子之前，但變數名稱之後卻不保證有一個賦值運算子。也就是說 parser 知道 A = 100 是個賦值作業，但也必須夠聰明地知道 A/10 不是賦值作業。為了達到目的，**evalExp1()**會從輸入流（input stream）中讀取後續的 token。如果那不是個 = 符號，就呼叫底下的 **putBack()**把 token 歸還給輸入流：

```
#001 // 把某個 token 歸還給輸入流（input stream）
#002 private void putBack()
#003 {
#004   if(token == EOE) return;
#005   for(int i=0; i < token.length(); i++) expIdx--;
#006 }
```

完成所有必要的變更後，parser 目前看起來像這樣：

```
#001 /*
#002    本模組包含一個可使用變數的遞迴漸降式 parser
#003 */
#004
#005 // 針對 parser 錯誤狀況而準備的異常類別（exception class）
#006 class ParserException extends Exception {
#007   String errStr; // 用以描述錯誤
#008
#009   public ParserException(String str) {
#010     errStr = str;
#011   }
#012
#013   public String toString() {
#014     return errStr;
#015   }
#016 }
#017
#018 class Parser {
#019   // 這些是 token 類型
#020   final int NONE = 0;
#021   final int DELIMITER = 1;
#022   final int VARIABLE = 2;
#023   final int NUMBER = 3;
#024
#025   // 這些是語法錯誤類型
#026   final int SYNTAX = 0;
#027   final int UNBALPARENS = 1;
#028   final int NOEXP = 2;
#029   final int DIVBYZERO = 3;
#030
#031   // 這個 token 代表「算式尾端」（end-of-expression）
#032   final String EOE = "\0";
#033
#034   private String exp;     // 指向算式字串
#035   private int expIdx;     // 算式的目前索引位置
#036   private String token;   // 用以儲存目前的 token
#037   private int tokType;    // 用以儲存 token 類型
#038
#039   // 陣列，供變數使用
#040   private double vars[] = new double[26];
#041
#042   // parser 進入點
#043   public double evaluate(String expstr) throws ParserException
#044   {
#045     double result;
#046     exp = expstr;
```

```
#047    expIdx = 0;
#048
#049    getToken();
#050    if(token.equals(EOE))
#051      handleErr(NOEXP); // 沒有看到任何算式
#052
#053    // 對算式進行解析與演算
#054    result = evalExp1();
#055
#056    if(!token.equals(EOE)) // 最後一個 token 必須是 EOE
#057      handleErr(SYNTAX);
#058
#059    return result;
#060  }
#061
#062  // 處理賦值（assignment）作業
#063  private double evalExp1() throws ParserException
#064  {
#065    double result;
#066    int varIdx;
#067    int ttokType;
#068    String temptoken;
#069
#070    if(tokType == VARIABLE) {
#071      // 儲存原來的 token
#072      temptoken = new String(token);
#073      ttokType = tokType;
#074
#075      // 計算變數的索引值
#076      varIdx = Character.toUpperCase(token.charAt(0)) - 'A';
#077
#078      getToken();
#079      if(!token.equals("=")) {
#080        putBack(); // 傳回目前的 token
#081        // 取回原來的 token – 不是賦值命令
#082        token = new String(temptoken);
#083        tokType = ttokType;
#084      }
#085      else {
#086        getToken(); // 取得 exp 的下一成分
#087        result = evalExp2();
#088        vars[varIdx] = result;
#089        return result;
#090      }
#091    }
#092
#093    return evalExp2();
#094  }
```

```
#095
#096   // 加或減兩個「項」（terms）
#097   private double evalExp2() throws ParserException
#098   {
#099     char op;
#100     double result;
#101     double partialResult;
#102
#103     result = evalExp3();
#104
#105     while((op = token.charAt(0)) == '+' || op == '-') {
#106       getToken();
#107       partialResult = evalExp3();
#108       switch(op) {
#109         case '-':
#110           result = result - partialResult;
#111           break;
#112         case '+':
#113           result = result + partialResult;
#114           break;
#115       }
#116     }
#117     return result;
#118   }
#119
#120   // 乘或除兩個「因子」（factor）
#121   private double evalExp3() throws ParserException
#122   {
#123     char op;
#124     double result;
#125     double partialResult;
#126
#127     result = evalExp4();
#128
#129     while((op = token.charAt(0)) == '*' ||
#130           op == '/' || op == '%') {
#131       getToken();
#132       partialResult = evalExp4();
#133       switch(op) {
#134         case '*':
#135           result = result * partialResult;
#136           break;
#137         case '/':
#138           if(partialResult == 0.0)
#139             handleErr(DIVBYZERO);
#140           result = result / partialResult;
#141           break;
#142         case '%':
#143           if(partialResult == 0.0)
```

```
#144            handleErr(DIVBYZERO);
#145          result = result % partialResult;
#146          break;
#147        }
#148      }
#149      return result;
#150    }
#151
#152    // 處理指數（exponent）
#153    private double evalExp4() throws ParserException
#154    {
#155      double result;
#156      double partialResult;
#157      double ex;
#158      int t;
#159
#160      result = evalExp5();
#161
#162      if(token.equals("^")) {
#163        getToken();
#164        partialResult = evalExp4();
#165        ex = result;
#166        if(partialResult == 0.0) {
#167          result = 1.0;
#168        } else
#169          for(t=(int)partialResult-1; t > 0; t--)
#170            result = result * ex;
#171      }
#172      return result;
#173    }
#174
#175    // 處理一元運算 + 或 -
#176    private double evalExp5() throws ParserException
#177    {
#178      double result;
#179      String  op;
#180
#181      op = "";
#182      if((tokType == DELIMITER) &&
#183         token.equals("+") || token.equals("-")) {
#184        op = token;
#185        getToken();
#186      }
#187      result = evalExp6();
#188
#189      if(op.equals("-")) result = -result;
#190
#191      return result;
#192    }
```

```
#193
#194   // 處理「括號內的算式」
#195   private double evalExp6() throws ParserException
#196   {
#197     double result;
#198
#199     if(token.equals("(")) {
#200       getToken();
#201       result = evalExp2();
#202       if(!token.equals(")"))
#203         handleErr(UNBALPARENS);
#204       getToken();
#205     }
#206     else result = atom();
#207
#208     return result;
#209   }
#210
#211   // 取得一個數字（number）的值（value）
#212   private double atom() throws ParserException
#213   {
#214     double result = 0.0;
#215
#216     switch(tokType) {
#217       case NUMBER:
#218         try {
#219           result = Double.parseDouble(token);
#220         } catch (NumberFormatException exc) {
#221           handleErr(SYNTAX);
#222         }
#223         getToken();
#224         break;
#225       case VARIABLE:
#226         result = findVar(token);
#227         getToken();
#228         break;
#229       default:
#230         handleErr(SYNTAX);
#231         break;
#232     }
#233     return result;
#234   }
#235
#236    // 傳回變數的值
#237   private double findVar(String vname) throws ParserException
#238   {
#239     if(!Character.isLetter(vname.charAt(0))){
#240       handleErr(SYNTAX);
#241       return 0.0;
```

```
#242    }
#243    return vars[Character.toUpperCase(vname.charAt(0))-'A'];
#244  }
#245
#246  // 把某個 token 歸還給輸入流（input stream）
#247  private void putBack()
#248  {
#249    if(token == EOE) return;
#250    for(int i=0; i < token.length(); i++) expIdx--;
#251  }
#252
#253  // 處理錯誤狀況
#254  private void handleErr(int error) throws ParserException
#255  {
#256    String[] err = {
#257      "Syntax Error",
#258      "Unbalanced Parentheses",
#259      "No Expression Present",
#260      "Division by Zero"
#261    };
#262
#263    throw new ParserException(err[error]);
#264  }
#265
#266  // 取下一個 token
#267  private void getToken()
#268  {
#269    tokType = NONE;
#270    token = "";
#271
#272    // 檢查是否為算式尾端
#273    if(expIdx == exp.length()) {
#274      token = EOE;
#275      return;
#276    }
#277
#278    // 跳過空白字元（white space）
#279    while(expIdx < exp.length() &&
#280      Character.isWhitespace(exp.charAt(expIdx))) ++expIdx;
#281
#282    // 檢查是否為算式尾端的空白字元
#283    if(expIdx == exp.length()) {
#284      token = EOE;
#285      return;
#286    }
#287
#288    if(isDelim(exp.charAt(expIdx))) { // 是個運算子
#289      token += exp.charAt(expIdx);
```

```
#290      expIdx++;
#291      tokType = DELIMITER;
#292    }
#293    else if(Character.isLetter(exp.charAt(expIdx))) { // 是個變數
#294      while(!isDelim(exp.charAt(expIdx))) {
#295        token += exp.charAt(expIdx);
#296        expIdx++;
#297        if(expIdx >= exp.length()) break;
#298      }
#299      tokType = VARIABLE;
#300    }
#301    else if(Character.isDigit(exp.charAt(expIdx))) { // 是個數字
#302      while(!isDelim(exp.charAt(expIdx))) {
#303        token += exp.charAt(expIdx);
#304        expIdx++;
#305        if(expIdx >= exp.length()) break;
#306      }
#307      tokType = NUMBER;
#308    }
#309    else { // 遇到未知字元便結束整個算式
#310      token = EOE;
#311      return;
#312    }
#313  }
#314
#315  // 如果 c 是個定義符號（delimiter），就傳回 true
#316  private boolean isDelim(char c)
#317  {
#318    if((" +-/*%^=()".indexOf(c) != -1))
#319      return true;
#320    return false;
#321  }
#322 }
```

要測試這個加強版的 parser，可使用先前測試簡易版 parser 的同一個程式。現在您可以輸入這樣的算式了：

```
A = 10/4
A - B
C = A * (F - 21)
```

2.7. 在遞迴漸降解析器中進行語法檢驗

Syntax Checking in a Recursive-Descent Parser

在算式解析過程當中，語法錯誤（syntax error）是指輸入的算式不符合 parser 的嚴謹規則。大多數時候屬於人為疏失，通常是打字上的錯誤。例如下面這些算式對本章的 parser 來說就是無效的式子：

```
10 ** 8
((10 - 5) * 9
/8
```

第一個式子中有兩個運算子連在一起，第二個式子帶有不對稱的括號，最後一個式子則是一開始就使用除法。這些情況都不是 parser 所允許的。由於語法錯誤可能導致 parser 送出錯誤的結果，因此您必須防範它們。

當 parser 偵測到錯誤狀況時，就呼叫 **handleErr()**。與其他類型的 parser 不同，遞迴漸降式 parser 使語法檢驗工作變得比較容易，因為大多數情況下，錯誤發生在 **atom()**、**findVar()**，或檢驗括號的 **evalExp6()**中。

一旦 **handleErr()**被呼叫，它會丟出一個 **ParserException**，內含錯誤描述。**Parser** 不會捕捉這個異常，而是把它丟往呼叫端。因此當遭遇錯誤時 parser 會立刻停止。當然，您可以根據自己的需要來變更這種行為。

您或許會想做一件事：展開 **ParserException** object 所含的資訊。以目前編寫的程式碼而言，此 class 只保存一個用以描述錯誤的字串。您或許會想把錯誤代碼、錯誤發生時算式字串的索引位置，或其他資訊加進去。

2.8. 一個計算器小程式

A Calculator Applet

這個 parser 用起來很簡單，而且幾乎可以被加到任何應用程式中。為瞭解這個 parser 的使用有多麼容易，試考慮以下範例。只要數行程式碼就建立出一個相當實用的計算器小程式（applet）。這個計算器程式用到兩個文字欄（text fields），其中第一個內含待被核算之算式，第二個用來顯示核算結果，那將會是個唯讀的文字欄。錯誤訊息顯示於狀態列（status line）。輸出畫面如圖 2-1（使用 Applet Viewer）。

```
#001 // 簡易計算機 applet
#002 import java.awt.*;
#003 import java.awt.event.*;
#004 import java.applet.*;
#005 /*
#006   <applet code="Calc" width=200 height=150>
#007   </applet>
#008 */
#009
#010 public class Calc extends Applet
#011   implements ActionListener {
#012
#013   TextField expText, resText;
#014   Parser p;
#015
#016   public void init() {
#017     Label heading = new
#018          Label("Expression Calculator ", Label.CENTER);
#019
#020     Label explab = new Label("Expression ", Label.CENTER);
#021     Label reslab = new Label("Result     ", Label.CENTER);
#022     expText = new TextField(24);
#023     resText = new TextField(24);
#024
#025     resText.setEditable(false); // 用以放置核算結果。只做顯示用。
#026
#027     add(heading);
#028     add(explab);
#029     add(expText);
#030     add(reslab);
#031     add(resText);
#032
#033     /* 將算式文字欄（text field）註冊，使得以接受動作事件（active events）
*/
#034     expText.addActionListener(this);
#035
```

```
#036     // 建立 parser
#037     p = new Parser();
#038   }
#039
#040   // 使用者按下 Enter 鍵
#041   public void actionPerformed(ActionEvent ae) {
#042     repaint();
#043   }
#044
#045   public void paint(Graphics g) {
#046     double result = 0.0;
#047     String expstr = expText.getText();
#048
#049     try {
#050       if(expstr.length() != 0)
#051         result = p.evaluate(expstr);
#052
#053 // 若要使 Enter 鍵被按下後便清除算式欄位，
#054 // 可使用下面這行程式碼：
#055 //    expText.setText("");
#056
#057       resText.setText(Double.toString(result));
#058
#059       showStatus(""); // 清除先前的任何錯誤訊息
#060     } catch (ParserException exc) {
#061       showStatus(exc.toString());
#062       resText.setText("");
#063     }
#064   }
#065 }
```

Calc 一開始就先宣告了三個實體變數（instance variables）。前兩個是 **expText** 和 **resText**，分別儲存指向「算式文字欄」和「結果文字欄」的 references。另一個「指向 parser」的 reference 則存放於變數 **p**。

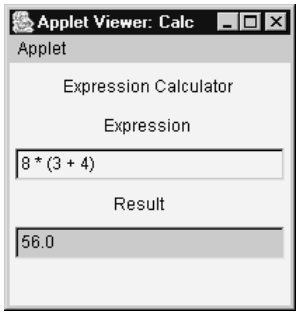

圖2-1 一個簡單而有效的計算器小程式（calculator applet）

在 **init()**中，文字欄被產生出來並加入 applet。每當用戶在文字欄按下 Enter，就會產出一個「動作事件」（action event）。由於儲存核算結果的文字欄 **resText** 只做為資料顯示之用，因此先呼叫 **setEditable(false)** 將它設為唯讀類型。這會使得它變暗，並且無法回應用戶的輸入。最後，程式會具現化一個 **Parser** 實體並指派給 **p**。

使用這個計算器時，可簡單輸入某個算式並按下 Enter 鍵。這會導致一個 **ActionEvent** 被產生出來，它由 **actionPerformed()**處理，並因而喚起 **repaint()**，最終導致 **paint()**被喚起。**paint()** 會執行 parser，核算算式，並顯示結果。若有錯誤，錯誤訊息會被顯示於狀態列（status line）。

2.9. 用 按用 亂

本章示範的算式解析器（expression parser）在各類應用中相當有價值，因為它使您在不花費太多功夫的情況下就可以為應用程式提供一些擴充功能。假如您的程式要求使用者輸入一個數值，例如某個應用程式可能要求使用者輸入文件列印的副本數。通常您會僅僅顯示一個文字欄，等候使用者輸入，然後把文字轉成內部數值格式。這種簡便處理手法允許使用者輸入一個像 100 這樣的數值。然而若使用者想為 9 個部門分別列印 72 份副本，會是怎樣的情況呢？使用者必須手動計算結果，再把結果 648 輸入文字欄中。如果您能利用 parser 核算文字欄內的資料，使用者就可以直接輸入 9*72，不需親自計算。一旦具備「數值算式」（numeric expression）的解析及核算能力，就可以為哪怕最簡單的應用程式添增不落俗套的專業感。是的，請試試在您的某個應用程式身上以 parser 處理數值輸入。

一如本章一開始所提，這個 parser 只進行最少量的錯誤檢驗工作。您或許會想加入更詳細的錯誤回報，例如把算式發生錯誤處強調出來，這就能夠讓使用者輕鬆找到語法錯誤並更正。

目前開發的這個 parser 只能針對數值算式（numeric expression）進行核算。只要加上一些東西，就可以讓它核算其他類型的算式，像字串、空間座標、複數等等。如果您要讓這個 parser 核算字串，必須為它做以下變更：

1. 定義一個稱為 **STRING** 的 token 新類型。

2. 加強 **getToken()**，使它能夠辨識字串。

3. 在 **atom()**中新增一個 **case**，以便處理 **STRING** 類型的 token。

實現這一切之後，parser 就能夠處理像下面這樣的字串算式（string expression）：

```
a = "one"
b = "two"
c = a + b
```

c 之中存放的應該是 **a** 和 **b** 連接起來的結果，也就是 "onetwo"。