

Java^{SE6}

技術
手冊

- 良葛格 Java 學習心得分享
- 新增 JDBC 介紹
- Java SE 6 新功能介紹
- 適用各領域的輸入輸出、執行緒、反射等主題



目錄

1. 關於 Java SE 6 技術手冊
2. 瞭解 Java
 - i. 什麼是 Java
 - ii. Java 的特性
 - iii. 如何學習 Java
 - iv. 接下來的主題
3. 入門準備
 - i. 下載、安裝、瞭解 JDK
 - ii. 設定 Path 與 Classpath
 - iii. 第一個 Java 程式
 - iv. 選擇開發工具
 - v. 接下來的主題
4. 語法入門
 - i. 第一個 Java 程式
 - ii. 在文字模式下與程式互動
 - iii. 資料、運算
 - iv. 流程控制
 - v. 接下來的主題
5. 從 autoboxing、unboxing 認識物件
 - i. 關於物件
 - ii. 自動裝箱、拆箱
 - iii. 接下來的主題
6. 陣列
 - i. 一維陣列、二維陣列
 - ii. 進階陣列觀念
 - iii. 接下來的主題
7. 字串
 - i. 認識字串
 - ii. 字串進階運用
 - iii. 接下來的主題
8. 封裝（Encapsulation）
 - i. 定義類別（Class）
 - ii. 關於方法
 - iii. 接下來的主題
9. 繼承（Inheritance）、多型（Polymorphism）
 - i. 繼承
 - ii. 多型（Polymorphism）
 - iii. 接下來的主題
10. 管理類別檔案
 - i. 內部類別
 - ii. package 與 import
 - iii. 接下來的主題
11. 例外處理（Exception Handling）
 - i. 例外處理入門
 - ii. 受檢例外（Checked Exception）、執行時期例外（Runtime Exception）
 - iii. throw、throws
 - iv. 例外的繼承架構
 - v. 斷言（Assertion）
 - vi. 接下來的主題

12. 列舉型態 (Enumerated Types)

- i. 常數設置與列舉型態
- ii. 定義列舉型態
- iii. 接下來的主題

13. 泛型

- i. 泛型入門
- ii. 泛型進階語法
- iii. 接下來的主題

Java SE 6 技術手冊

這邊的電子書版本，是我玩 GitBook 的試作品，就一個線上電子書編輯器來說，GitBook 算是很不錯的選擇，不過自由度還比不上我目前做電子書的方式：

- [Google Play 電子書新手上架（一）從 MD 到 PDF、ePub](#)
- [Google Play 電子書新手上架（二）上架 PDF 與 ePub 檔案](#)

如果你對如何使用 GitBook 有興趣，可以參考〈[深入淺出 GitBook 寫作與自助出版，電子書也能多人協作](#)〉。

這邊我試排了《[Java SE 6 技術手冊](#)》的半本，你可以藉此看看 GitBook 的效果，並與 [Google Play 上的版本](#) 對照看看效果！

如果你需要其他技術文件，可以在 [我的網站](#) 上找到。

第 1 章 瞭解 Java

如果您完全沒有接觸過 Java 或是僅對 Java 有著模糊的認識，那麼試著在這一章中，從 10 多年以來，各時期的 Java 所擔任的角色來瞭解它，或是從 Java 的語言特色來瞭解它、從 Java 應用的平台特色來瞭解它，以及從各式各樣活躍的 Java 社群來瞭解它。如果您是 Java 的初學者，我也在這章最後提示了一些如何學好 Java 的建議。

這一章完全是簡介性的內容，對往後的學習不會有什麼影響，如果您想馬上開始學習 Java，則可以先跳過這個章節，待日後有空時再回過頭來看看這個章節的內容。

什麼是 Java

1.1 什麼是 Java

在不同的時間點上，Java 這個名詞有著不同的意義，要瞭解什麼是 Java，從一些歷史性的資料上，您可以對 Java 的角色有所瞭解。

Java 最早是 Sun 公司（Sun Microsystems Inc.）「綠色專案」（Green Project）中撰寫 Star7 應用程式的一個程式語言，當時的名稱不是 Java，而是取名為 Oak。

綠色專案開始於 1990 年 12 月，由 Patrick Naughton、Mike Sheridan 與 James Gosling 主持，目的是希望構築出下一波電腦應用的趨勢並加以掌握，他們認為下一波電腦應用的趨勢將會集中在消費性數位產品（像是今日的 PDA、手機等消費性電子商品）的使用上，在 1992 年 9 月 3 日 Green Team 專案小組展示了 Star7 手持設備，這個設備具備了無線網路連接、5 吋的 LCD 彩色螢幕、PCMCIA 介面等功能，而 Oak 在綠色專案中的目的，是用來撰寫 Star7 上應用程式的程式語言。

Oak 名稱的由來，是因為 James Gosling 的辦公室窗外有一顆橡樹（Oak），就順手取了這個名稱，但後來發現 Oak 名稱已經被註冊了，工程師們邊喝咖啡邊討論著新的名稱，最後靈機一動而改名為您所常聽到的 Java。

全球資訊網（World Wide Web）興起，Java Applet 成為網頁互動技術的代表。

1993 年第一個全球資訊網瀏覽器 Mosaic 誕生，James Gosling 認為網際網路與 Java 的一些特性不謀而合，利用 Java Applet 在瀏覽器上展現互動性的媒體，在當時而言，對人們的視覺感官是一種革命性的顛覆，Green Team 仿照了 Mosaic 開發出一個以 Java 技術為基礎的瀏覽器 WebRunner（原命名為 BladeRunner），後來改名為 HotJava，雖然 HotJava 只是一個展示性的產品，但它使用 Java Applet 展現的多媒體效果馬上吸引許多人的注意。

1995 年 5 月 23 日，Java Development Kits（當時的 JDK 全名）1.0a2 版本正式對外發表，而在 1996 年 Netscape Navigator 2.0 也正式支援 Java，Microsoft Explorer 亦開始支援 Java，從此 Java 在網際網路的世界中逐漸風行起來，雖然 Star7 產品並不被當時的消費性市場所接受，綠色專案面臨被裁撤的命運，然而全球資訊網（World Wide Web）的興起卻給了 Java 新的生命與舞台。

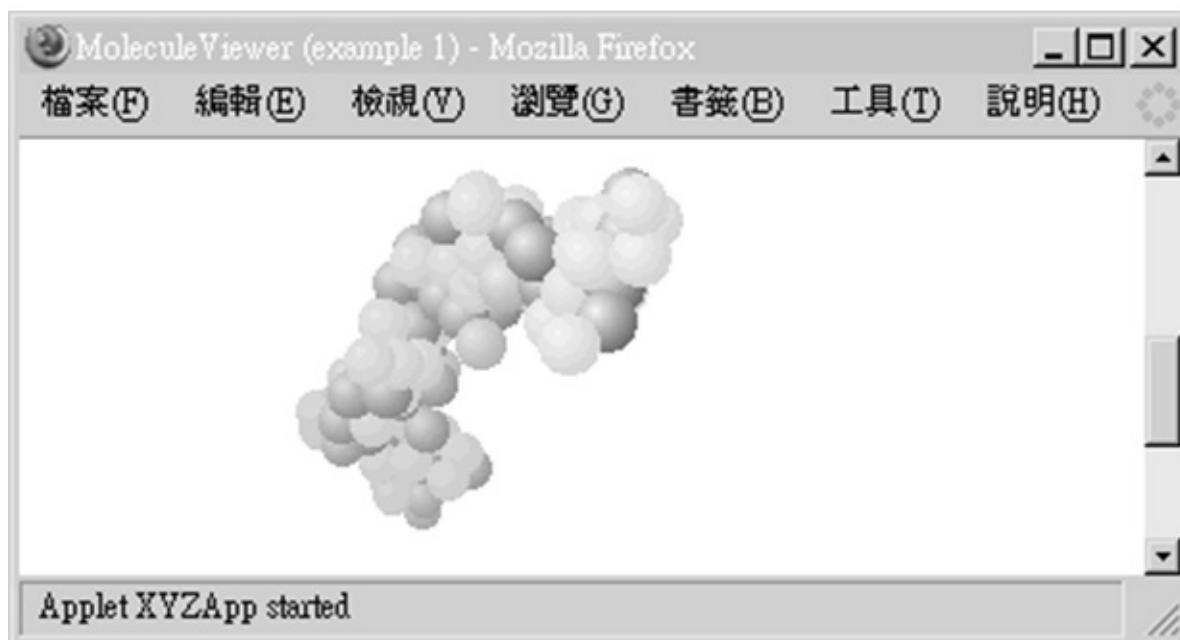


圖1.1. JDK 所附的 Java Applet 範例（jdk 目錄\demo\applets\MoleculeViewer\ example1.html）

Java 是一個更簡單的物件導向（Object-Oriented）程式語言，具有更高的跨平台可能性。Java 是一個支援物件導向觀念的程式語言，在使用上比 C++ 更為簡單，它限制或簡化了 C++ 語言在開發程式時的一些功能，雖然犧牲了某些存取或彈性，但讓開發人員避免開發軟體時可能發生的錯誤，並讓程式語言本身使用上更為方便，而 Java 所撰寫出來的程式在不同的平台間具有更高的可攜性，對於「撰寫一次，到處執行」（Write Once, Run Anywhere）這樣的夢想，Java 提供了更高的可能性。

Java可以代表程式語言，但在今日，更多時候代表了軟體開發的架構。

Java 的開發者版本在發表時是以 Java Development Kits 名稱發表，簡稱 JDK，到 J2SE 5.0 (Java 2 Platform Standard Edition 5.0) 時的 JDK，稱為 J2SE Development Kit 5.0，從 Java SE 6 (Java Platform, Standard Edition 6) 開始的 JDK6 則稱之為 Java SE Development Kit 6，也就是不再像以前 Java 2 帶有 "2" 這個號碼，版本號 6 或 1.6.0 都使用，6 是產品版本 (product version)，而 1.6.0 是開發者版本 (developer version)。

JDK 除了提供撰寫Java程式時所必要的編譯、執行、除錯等工具之外，更搭配有越來越豐富的 API (Application Programming Interface)，隨著應用範圍的越來越廣，Java 演化出三個不同領域的應用平台：**Java SE**、**Java EE** 與 **Java ME**（在這之前的舊名稱是 J2SE、J2EE 與 J2ME）。

Java 不再只是單純的程式語言加上 API 文件庫的組成，更提供開發人員在各個領域開發軟體時，一種依循的標準與框架（Framework）工具。總而言之，隨著時間的推演，Java 這個名詞不再只是表示一個程式語言，而是一種開發軟體的平台，更進一步的也是一種開發軟體時的標準與架構的統稱，事實上語言在整個 Java 的藍圖中只不過是一個極小的部份，學習 Java 本身也不僅僅在於學習如何使用它的語法，更多的時候是在學習如何應用Java所提供的資源與各種標準，以開發出架構更好、更容易維護的軟體。

良葛格的話匣子 在 Design Patterns Elements of Reusable Object-Oriented Software 書中對「框架」作出的解釋是：框架就是一組互相合作的類別組成，它們為特定類型的軟體開發提供了一個可以重複使用的設計。

1.2 Java 的特性

Java 本身是個程式語言，所以您可以從程式語言本身的特性來討論它，Java 擁有不同領域的平台，所以您可以從應用領域與平台的特性來探討它，更重要的是 Java 擁有許多活躍的社群、資源與開放原始碼（Open source）專案，這更是在討論 Java 時值得一提的特性。

1.2.1 語言特性

作為一個程式語言，Java 擁有許多重要的特性：簡單的（Simple）、物件導向的（Object-oriented）、網路的（Network-savvy）、解譯的（Interpreted）、堅固的（Robust）、安全的（Secure）、可攜的（Portable）、高效能的（High-performance）。以下我針對這幾個重要的特性來加以說明。

- 簡單的（Simple）

C/C++ 的功能強大是大家所皆知的，即使在眾多程式語言的競爭之下，C/C++ 仍舊在開發軟體的程式語言間佔有相當的地位，然而學習或使用 C/C++ 並不容易，很多時候開發人員並不需要使用到 C/C++ 的一些功能，但為了使用 C/C++ 却得付出相當的學習成本、開發成本或維護成本。

設計 Java 的成員們以長年的開發經驗判斷，在使用一些 C/C++ 的功能時，所得到的壞處可能多於好處（尤其是被一些沒有經驗的開發人員使用時），所以 Java 捨棄了 C/C++ 中一些較少使用、難以掌握或可能不安全的功能，像是指標（Pointer）、運算子重載（Operator overloading）、多重繼承（Multiple inheritance）等等。

Java 除去一些 C/C++ 複雜或不安全的功能，並在許多開發人員常使用的特性上加以簡化而使之易於使用，例如字串在 Java 中的處理就更為簡單；Java 在設計時參考了許多 C/C++ 的語法與特性，使得學習過 C/C++ 的開發人員可以在短時間內瞭解如何使用 Java。

- 物件導向的（Object-oriented）

物件導向分析（Object-oriented Analysis）是分析問題的一種方式，物件導向設計（Object-oriented Design）是使用物件導向的思考方式來設計問題的解決方案。很不幸的，要瞭解什麼是物件導向，以及學會使用物件導向的方式進行設計與解決問題，並不是一件簡單的事情，事實上讓您瞭解物件導向正是這本書的目標之一。

這邊您先不用急於瞭解什麼是物件導向，在往後的章節中我會逐漸以實際的例子讓您體會什麼是物件導向，現階段您所要瞭解的是，Java 支援物件導向的設計方式，簡單的說，Java 讓您可以用物件導向的思考與方式來設計並撰寫程式，物件導向的好處之一，就是可以讓您設計出可重用的元件，或者是直接使用別人所撰寫好的 Java 元件，並使開發出來的軟體更具彈性且容易維護。

- 網路的（Network-savvy）

Java 本身等於就是藉由網路而重生的，它的許多功能與應用都與網路相關，從最初的 Applet、簡化的 Socket、互動式的 JSP/Servlet 網路程式到今日熱門的 Web Service 等，都註定了 Java 在網路相關的領域佔有一席之地，事實上 Java 應用最多的領域也正是網路服務這一塊領域。

- 解譯的（Interpreted）

您要先知道一件事，程式在一個平台上要能夠執行，必須先編譯為該平台看得懂的原始機器指令（Native machine instructions），但最大的問題在於每個平台所認識的機器指令各不相同，例如 Windows 作業系統認識的機器指令就與 Linux 認識的不相同，專為 Windows 作業系統所撰寫並編譯好的程式，並無法直接拿來在 Linux 作業系統上執行。

為了解決不同平台間執行程式的問題，Java 的程式在進行編譯時，並不直接編譯為與平台相依的原始機器指令，而是編譯為與系統無關的「位元碼」（bytecodes），為了要執行 Java 程式，執行的平台上必須安裝有 JVM（Java Virtual Machine），JVM 就是 Java 位元碼檔案的虛擬作業系統，Java 位元碼檔案就是 JVM 的可執行檔案，當運行 Java 程式時，Java 即時編譯器（Just In Time compiler, JIT）會將位元碼解譯為目標平台所認得的原始機器指令，藉由 JVM 使得

Java 程式在不同平台上都能執行的目的得以實現。

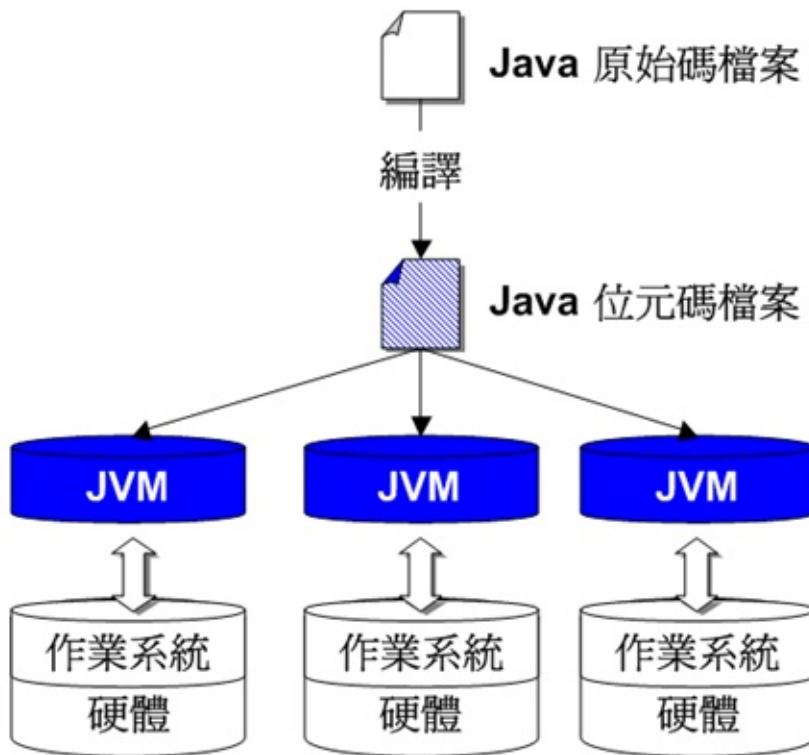


圖1.2. JVM 將 Java 位元碼轉換為平台相依的機器指令

良葛格的話匣子「平台」（Platform）一詞並沒有較嚴謹的定義，在電腦的領域中，平台有時指的是硬體，有時指的是作用於硬體之上的軟體系統，有時則指硬體加上軟體，這邊指的平台是「作業系統」，由於不同的硬體會運行不同的作業系統，所以這邊對平台的定義也就是硬體加上軟體。您可以在 [Google](#) 上搜尋 define:platform 來查詢網路上各種對平台的定義。

- 堅固的（Robust）

Java 將 C/C++ 中一些功能強大但不容易掌握的功能去除掉，以指標（Point）功能為例，即使是有經驗的開發人員在使用指標功能時也得小心翼翼，避免撰寫出使程式崩潰（Crash）的錯誤，諸如此類的功能在 Java 中被去除掉，為的是讓 Java 在使用時更為簡單，撰寫出來的程式更為堅固（Robust）。

捨棄了 C/C++ 的一些特性會使得許多開發人員質疑沒有了 C/C++ 這些特性，Java 還能開發什麼軟體？但從 Java 正式提出至今日10年來已經證明，Java 確實適用於開發各個領域的軟體，而且更擁有廣大的網路資源。

- 安全的（Secure）

Java 加入了自動垃圾收集（Garbage collection），讓開發人員無需擔心物件資源的回收問題，例外（Exception）處理架構讓開發人員可以掌握程式中各種突發的例外狀況，另外像是 synchronized、final 等存取關鍵詞的使用，目的都在於加強 Java 程式的安全性。

- 可攜的（Portable）

要讓程式跨平台並不是一件容易的事，有相當多的東西必須考量，例如資料型態所佔記憶體長度就是一個問題，Java 在不同的平台上之資料型態長度是統一的（而 C/C++ 則不然），這是 Java 在提高程式可攜性上最顯而易見的一個例子。

- 高效能的（High-performance）

高效能是 Java 所宣稱的，在某些條件的配合下，Java 號稱可以與 C/C++ 擁有同樣甚至更好的執行效能，但更多的人只

是將這個宣稱視為一個商業宣傳的口號，Java 是在執行時期才將中介的位元碼轉譯為原始機器碼，這就花上了一段不短的啟動時間，而早期的 Java 在執行效能上確實是一直被詬病的。

不過在歷經數個版本的變更，Java 一直嘗試提高其執行的效能，像是使用 Java HotSpot 技術，在第一次載入 Java 位元碼時，JIT 會以解譯模式開始載入，分析並嘗試以最佳化編譯為目標平台的原始機器碼。每一次的版本更新，Java 皆嘗試在效能上作出改進。

1.2.2 應用平台

Java 至今日主要發展出三個領域的應用平台：Java Platform, Standard Edition (Java SE)、Java Platform, Enterprise Edition (Java EE) 與 Java Platform, Micro Edition (Java ME)。

- Java Platform, Standard Edition (Java SE)

Java SE 是 Java 各應用平台的基礎，想要學習其它的平台應用，必先瞭解 Java SE 以奠定基礎，Java SE 也正是本書主要的介紹對象。

下圖是整個 Java SE 的組成概念圖：



圖1.3. Java SE 的組成概念圖

Java SE 可以分作四個主要的部份：**JVM、JRE、JDK 與 Java 語言**。

為了要能運行 Java 所撰寫好的程式，您的平台上必須有 Java 虛擬機器 (Java Virtual Machine, JVM)。JVM 包括在 Java 執行環境 (Java SE Runtime Environment, JRE) 中，所以為了要運行 Java 程式，您必須取得 JRE 並進行安裝。如果您要開發 Java 程式，則您必須取得 JDK (Java SE Development Kits)，JDK 包括了 JRE 以及開發過程中所需要的一些工具程式，像是 javac、java、appletviewer 等工具程式（關於 JRE 及 JDK 的安裝與使用介紹，會在第 2 章說明）。

Java 語言只是 Java SE 的一部份，除了語言之外，Java 最重要的就是它提供了龐大且功能強大的 API 類別庫，提供像是字串處理、資料輸入輸出、網路套件、使用者視窗介面等功能，您可以使用這些 API 作為基礎來進行程式的開發，而無須重複開發功能相同的元件，事實上，在熟悉 Java 語言之後，更多的時候，您都是在學習如何使用 Java SE 所提供的 API 來組成您的程式。

- Java Platform, Enterprise Edition (Java EE)

隨著 Java 的應用領域越來越廣，並逐漸擴及至各級應用軟體的開發，Sun 公司在 1999 年 6 月美國舊金山的 Java One 大會上，公佈了新的 Java 體系架構，該架構根據不同級別的應用開發區分了不同的應用版本：J2SE、J2EE 與 J2ME。這些是當時的名稱，為配合 Java SE 6 中的名稱，以下則稱為 Java SE、JavaEE 與 Java ME。

Java EE 以 Java SE 的基礎，定義了一系列的服務、API、協定等，適用於開發分散式、多層式（Multi-tiered）、以元件為基礎、以 Web 為基礎的應用程式，整個 Java EE 的體系是相當龐大的，比較為人所熟悉的技術像是 JSP、Servlet、Enterprise JavaBeans (EJB)、Java Remote Method Invocation (RMI) 等，當中的每個服務或技術都可以使用專書進行說明，所以並不是本書說明的範圍，但可以肯定的是，您必須在 Java SE 上奠定良好的基礎再來學習 Java EE 的開發。

- Java Platform, Micro Edition (Java ME)

Java ME 是 Java 平台版本中最小的一個，目的是作為小型數位設備上開發及部署應用程式的平台，像是消費性電子產品或嵌入式系統等，最為人所熟悉的設備如手機、PDA、股票機等，在近幾年已經相當常見 Java ME 的應用，越來越多的手持裝置都支援 Java ME 所開發出來的程式，像是 Java 遊戲、股票相關程式、記事程式、月曆程式等。

良葛格的話匣子 Java SE 6？JDK6？JRE6？您已經搞不清楚這些名稱了嗎？這邊再做個整理。Java SE 是指平台名稱，全名 Java Platform, Standard Edition 6。JDK6 是基於平台的開發程式發行版本，全名 Java SE Development Kit 6。JRE6 則是基於平台的執行環境發行版，全名 Java SE Runtime Environment 6。

1.2.3 活躍的社群與豐富的資源

Java 發展至今日獲得廣大開發者的支援，有一個不得不提的特性，即 Java 所擁有的各種豐富資源與各式活躍的社群，來自各個領域的開發人員與大師們各自對 Java 作出了貢獻。

無論是開發工具、開放原始碼的元件、Web 容器、測試工具、各式各樣的軟件專案、各個社群所支持的討論區、取之不盡的文件等，這些資源來自於各個商業化或非商業化的團體，各式各樣活躍的社群造就 Java 無限的資源，這些資源不僅具有實質的應用價值，更具有教育的價值，例如各式各樣的開放原始碼框架（Framework）成品，不僅可以讓您將之使用於實際的產品開發上，還可以讓您從中學習框架的架構與運行機制，即使在某些產品開發上您不使用 Java 來開發程式，也可以運用到這些框架的架構與運行機制。

1.3 如何學習 Java

如果您是 Java 的初學者，最想要知道的莫過於如何才能學好 Java？以下是我的幾點建議。

- 奠定 Java 語法基礎

學習 Java 的第一步，就是學會使用 Java 這個程式語言來撰寫程式，而學習程式語言的第一步，就是熟悉其語法的使用，程式語言就是一門語言，所不同的是這種語言是用來與電腦溝通的，所以要熟悉語言的話，使用的方法莫過於多觀摩別人寫的程式，瞭解別人是如何使用 Java 來解決他們的問題，然後針對同樣的程式進行練習，並從實作中測試自己是否真正瞭解到如何解決問題。

- 運用基本的 Java SE API

除了 Java 語言本身的語法之外，懂得運用 Java SE 的 API 也是一個必要的課題，然而在這麼多的 API 下，您必然想知道哪些API是必要或常用的，我的建議是先掌握字串處理、例外處理、物件容器（Container）、輸入輸出（I/O）、執行緒（Thread）這幾個主題。

API 的內容龐大，沒有任何一本書可以詳細講解每個 API 如何運用，您也不需要將 API 背誦下來，您要懂得查詢 API 文件說明，雖然 API 文件都是英文的，但基本上只要有基本的英文閱讀能力就足以應付查詢需求，以 Java SE 6 來說，您可以下面的網址查詢到 API 文件說明：

<http://java.sun.com/javase/6/docs/api/index.html>。

- 使用搜尋引擎

作為一個開發人員，懂得使用搜尋引擎來找尋問題的解答是一件必要的能力，我習慣使用 [Google](#)，幾個簡單的關鍵字通常就可以為您找到問題的答案。

- 加入社群參與討論

在學習的過程中，如果有人可以共同討論的話，將會加速您學習的速度，您可以找一個討論區並摸索當中的資源，這可以省去您不少的學習時間，在中文討論區中我建議多參與 [Java技術論壇](#)，論壇上有相當豐富的資源，您可以從「新手版 FAQ 目錄」開始，並記得在發問之前多使用「全文檢索」功能，搜尋論壇上是否有類似的討論。

- 學習地圖

在學習完基本的 Java SE 之後，您會想要實際應用Java來撰寫程式，如果您需要撰寫視窗程式，可以學習 Swing 視窗設計；如果您要撰寫資料庫相關軟體，可以學習 JDBC；如果您想要朝 Web 程式發展，可以學習 JSP/Servlet，如果您想要學習手機程式開發，可以朝 Java ME 方向學習。

在 Java 的官方網站上，有一篇 Java 技術概念地圖（Java Technology Concept Map），當中以圖表的方式描繪出各種需求下的學習方向參考，您可以瀏覽該圖表來瞭解 Java 各個技術主題之間有什麼關聯，以評估您未來學習的方向，Java 技術概念地圖的網址是：

<http://java.sun.com/developer/onlineTraining/new2java/javamap/intro.html>。

1.4 接下來的主題

每一個章節的內容由淺至深，初學者該掌握的深度要到哪呢？在這個章節中，對於初學者我建議至少掌握以下幾點內容：

- 什麼是位元碼（bytecode）
- Java 可以跨平台的原因
- Java 的三個應用領域（平台）
- Java SE、JDK、JRE、JVM 的差異

我喜歡從實作中學習新的事物並體驗當中運行的原理，並認為這是一個良好的學習方式，這種方式可以同時獲得觀念與實證。我也建議您從實作中學習，所以接下來我會以實作方式來介紹 Java 的每一個環節，首先第一步就是準備好撰寫 Java 程式的環境，包括安裝 JDK、設定環境變數、瞭解主控台（Console）操作以及選擇一個好的編輯器或整合開發環境（Integrated Development Environment）。

第 2 章 入門準備

Java 初學者最常遇到的狀況是 ... 在高高興興的下載了所謂的「Java 程式」安裝之後，就開始遇到一大堆的問題與挫折。下載、安裝程式從操作上看確實是很簡單，但是您真的知道下載了什麼東西嗎？您安裝了什麼程式？程式安裝到哪裡去？安裝完畢後該進行的相關設定又有哪些？為什麼要作這些設定？

如果您曾經因為安裝所謂的「Java 程式」而遇到一大堆的問題與挫折，您可以看看這個章節，重新瞭解您所要安裝的東西是什麼？瞭解什麼是 JDK、JRE？如何設定 Path 與 Classpath？如何成功撰寫第一個 Java 程式？

使用純文字檔案撰寫 Java 程式是第一步，建議每個 Java 的初學者從使用純文字檔案開始，然而所謂「工欲善其事、必先利其器」，選擇一個好的開發工具也是必要的，在這個章節的最後談到了一些開發工具，作為您選擇開發工具的參考。

2.1 下載、安裝、瞭解 JDK

要使用 Java 開發程式的第一步，就是安裝 JDK（Java SE Development Kit），這邊以 Java SE 6 Development Kit 安裝作為範例，以實作的方式一步步帶您瞭解 JDK。

2.1.1 下載 JDK

安裝 JDK 的第一步是先下載安裝檔案，這邊要下載的是 Sun 公司的 Java SE 6 Development Kit，下載的網址是：

- <http://java.sun.com/javase/downloads/index.jsp>

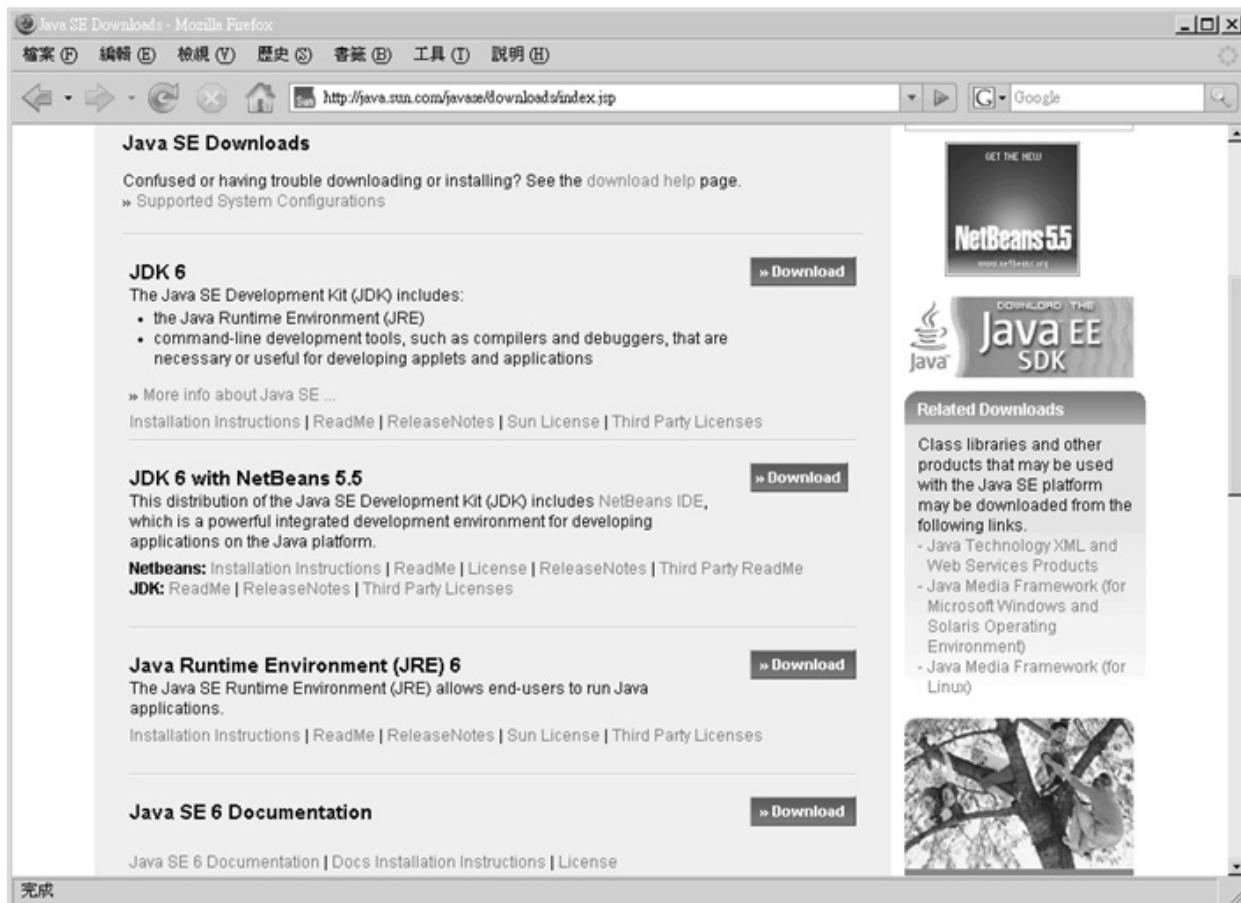


圖 2.1 Java SE 6 下載頁面

圖 2.1 是撰寫此書時 Java SE 6 下載網頁的擷取畫面，JDK 6 發表日期為 2006 年 12 月 11 日，Sun 公司會在必要的時候公佈修正版本以修正一些臭蟲（bug），您在看此書時可能已經有較新的版本，較新的修正版本將以 Update 名稱，加上號碼來表示修正的版本號。以下簡介頁面中的幾個可下載檔案之作用：

- JDK 6

JDK 6 是建議您要下載的安裝程式，這個下載檔案中包括了公用（Public）JRE 安裝選擇，所以您不必額外下載 JRE 6。

- JDK 6 with NetBeans 5.5

包括了 JDK 6 與 NetBeans 整合開發環境（Integrated Development Environment, IDE），有相當多的開發人員使用 NetBeans 來開發 Java 程式，這邊假設您沒有使用過任何的 IDE，所以不下載 JDK 6 with NetBeans 5.5 來安裝。

- Java Runtime Environment (JRE) 6

如果想要執行 Java 所開發的程式，電腦上必須安裝有 JRE，如果您打算讓其他人可以執行您所撰寫 Java 程式，則可以建議他下載 JRE 進行安裝。

- Java SE 6 Documentation

這份文件提供 Java SE 6 所有 API (Application Programming Interface) 的用途與使用方式介紹，在往後開發 Java 程式的過程中，會很頻繁的使用這份文件，您可以下載這份文件，往後就不必連上網路來查詢文件。除了以上幾個下載檔案的介紹之外，下載頁面上還有 JDK 6 的原始碼下載，以及一些進階開發人員才會用的到的下載檔案，目前您無需理會它們，您只要下載 JDK 6 就可以了。

以圖 2.1 為例，按下「JDK 6」的「Download」鏈結文字後，會連到「Download Center - Download」網頁，由於這邊的 JDK 6 是 Sun 公司的產品，所以您必須同意它的使用條款才可以下載，選擇「Accept」後，就可以進行 JDK 6 安裝檔案的下載。

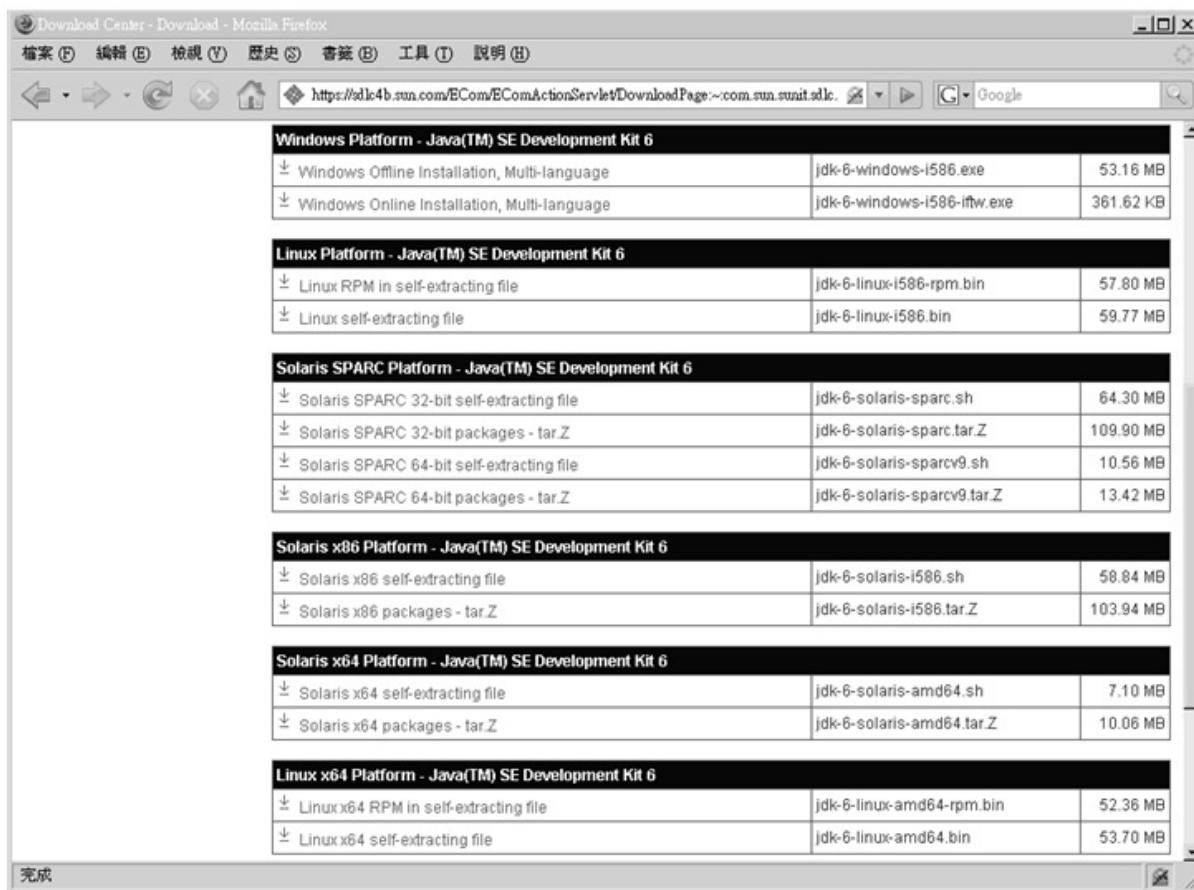


圖 2.2 Download Center – Download 網頁

在圖 2.2 中，您可以看到數個不同平台下的 JDK 安裝檔案，這邊假設您的作業系統是 Windows 2000/XP，所以請下載 Windows 平台的安裝檔案。檔案有 Windows Offline Installation, Multi-language，這個檔案是離線安裝檔案，所有安裝過程中必要的內容都在檔案中，所以檔案容量有 53 MB 之多，另外還有 Windows Online Installation, Multi-language，可以讓您根據實際選擇的項目透過網路進行安裝，如果您的網路速度夠快，可以下載這個檔案。

建議選擇下載離線安裝檔案，在安裝完畢後可以備份在電腦中，日後如果需要重新安裝的話就很方便。

2.1.2 安裝JDK

這邊假設您下載後的 JDK 安裝檔案名稱是 jdk-6-windows-i586.exe，按兩下這個檔案可以開始程式的安裝，開始的第一步是同意使用條款，再來則是開始安裝 JDK。

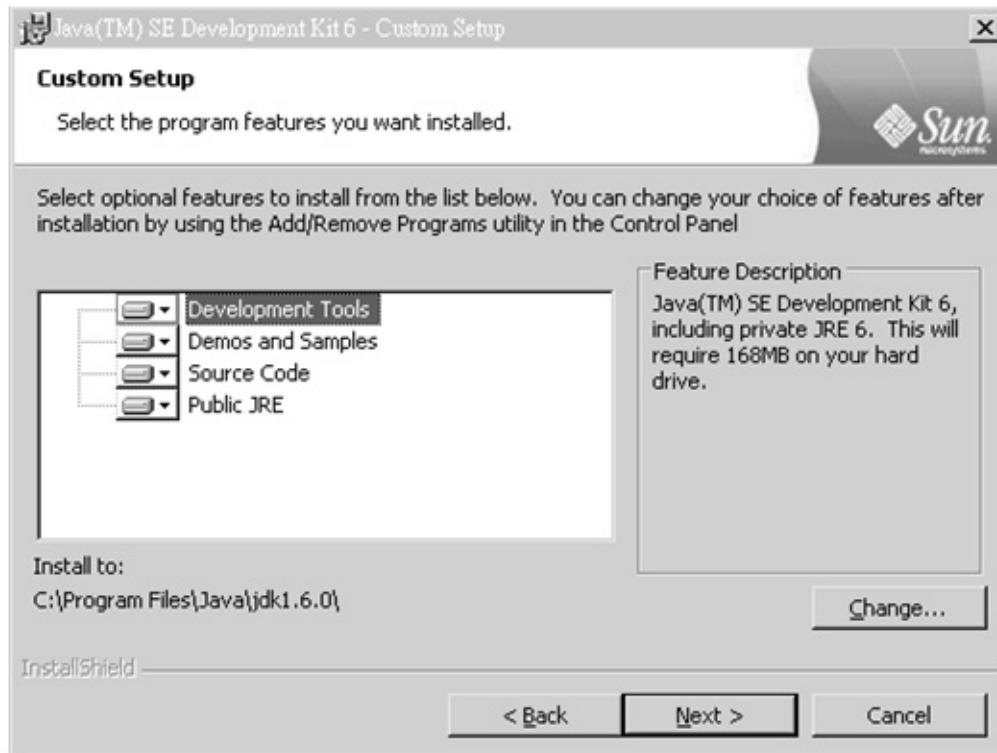


圖2.3 安裝 JDK 的畫面

在安裝 JDK 時可以讓您選擇安裝的項目，依序是開發工具（Development Tools）、示範程式（Demos）、API原始碼（Source Code）與公用 JRE（Public JRE）。開發工具是必需的，範例程式可供您日後寫作程式的參考，API 原始碼可以讓您瞭解所使用的 API 實際上是如何撰寫的，而 JRE 則是執行 Java 程式所必要的，所以這四個項目基本上都必須安裝。

要注意的是圖 2.3下面的「Install to」，請記下 JDK 安裝的位置，預設是「C:\Program Files\Java\jdk1.6.0」，您待會需要使用到這個資訊，如果想改變安裝目的地，可以按下「Change」按鈕來變更；接著按「Next」就開始進行 JDK 的安裝，完成 JDK 的安裝之後，接著會安裝「公用 JRE」。

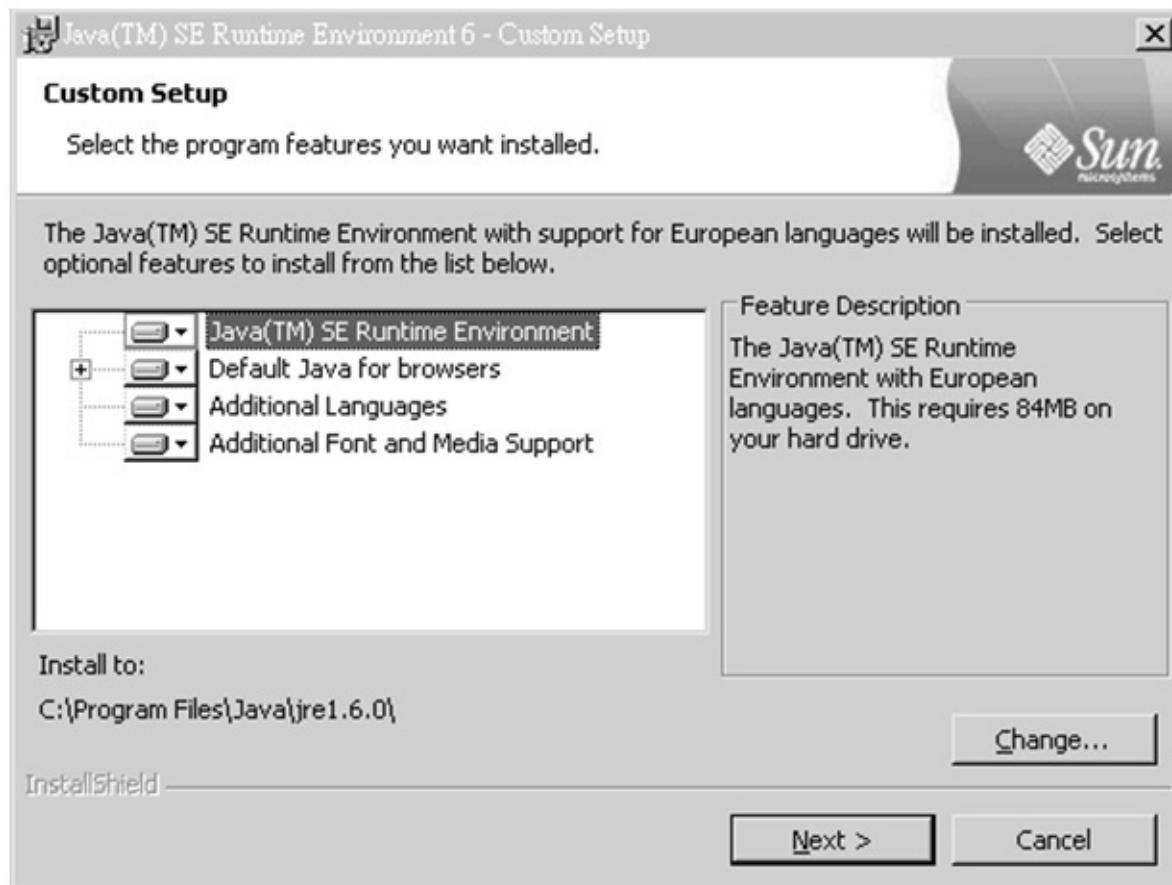


圖 2.4 安裝 JRE 的畫面

同樣的請留意圖 2.4 下方的「Install to」，瞭解 JRE 的安裝目的地，預設是「C:\Program Files\Java\jre1.6.0」，按下「Next」按鈕之後，會詢問哪些瀏覽器要使用 Java Plug-In？這可以讓您的瀏覽器可以執行 Java Applet，選擇要支援 Java Plug-In 的瀏覽器之後，按下「Next」鈕可以開始安裝公用 JRE。

2.1.3 瞭解 JDK

接著來瞭解一下您安裝的東西有哪些，這邊假設您的 JDK 與公用 JRE 各安裝至「C:\Program Files\Java\jdk1.6.0」及「C:\Program Files\Java\jre1.6.0」。

公用 JRE 主要是給開發好的 Java 程式執行的平台，之前曾經說過，JDK 本身也有自己的 JRE，這個 JRE 是位於JDK安裝目錄的「jre」目錄下，以之前的安裝為例，就是在「C:\Program Files\Java\jdk1.6.0\jre」中，JDK 本身所附的JRE主要是開發 Java 程式時測試之用，與公用 JRE 的主要差別，就在於JDK本身所附的 JRE 比公用 JRE 多了個 server 的 VM (Virtual Machine) 執行選項，您可以看看「C:\Program Files\Java\jdk1.6.0\jre\bin」與「C:\Program Files\Java\jre1.6.0\bin」就可以瞭解。



圖 2.5 JDK 的 JRE 有 server 選項



圖 2.6 公用 JRE 沒有 server 選項

server 與 client 選項的差別在於所使用的 VM 不同，執行 Java 程式時預設會使用 client VM，使用 server VM 的話會花比較長的啟動時間及耗用較多的記憶體，為的是啟動 Java 程式後可以獲得較好的執行效能；初學者現階段不用在乎啟動 server 或 client VM 的差別，只要先知道有這兩種 VM 的存在即可。

繼續來看到 JDK 的安裝目錄下有哪些東西。

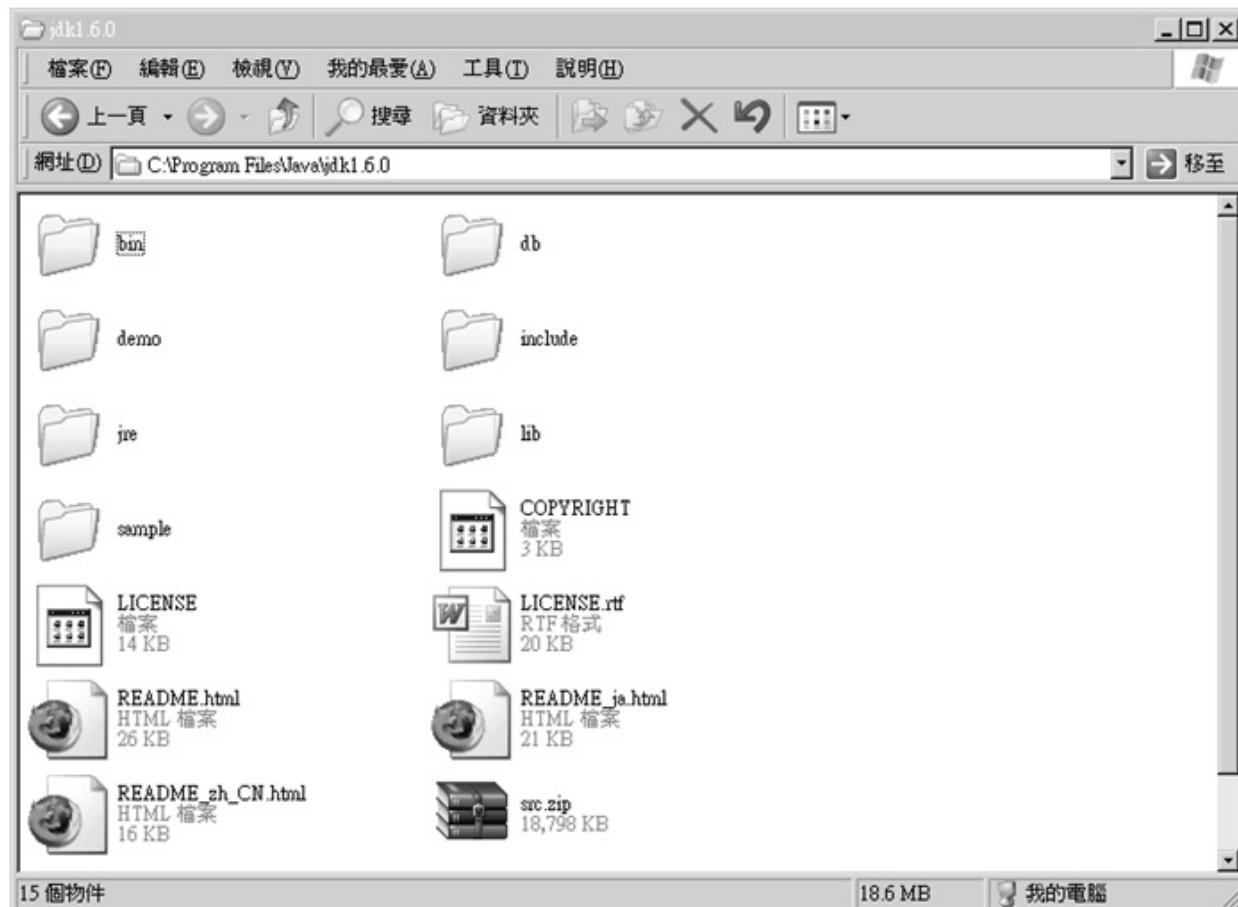


圖 2.7 JDK 安裝目錄的內容

- 「bin」 目錄

提供的是 JDK 的工具程式，包括了 javac、java、javadoc、appletviewer 等程式。

- 「demo」 目錄

一些使用 Java 撰寫好的範例程式。

- 「jre」 目錄

JDK自己附帶的JRE。

- 「db」 目錄

在 JDK 6 中所附帶的 Apache Derby 資料庫，這是純 Java 所撰寫的資料庫，支援 JDBC 4.0，在第 21 章有關於 Apache Derby 資料庫的一些簡介。

- 「lib」 目錄

工具程式實際上會使用的 Java 工具類別。JDK 中的工具程式，大多也是由 Java 所撰寫而成，bin 資料夾下的工具程式，例如 Windows 版本的 JDK 中，javac.exe、java.exe 等工具程式，它們不過是個包裝器（Wrapper），執行 javac.exe 等程式時，最後會呼叫 lib 目錄中 tools.jar 中的對應類別，例如 javac 工具程式實際上會去使用 tools.jar 中的 com/sun/tools/javac/Main 類別。

- src.zip

Java 提供的 API 類別之原始碼檔案壓縮檔，如果您將來有需要看看 API 的某些功能是如何實作的，可以看看這個檔案

中的原始碼內容。

在大致瞭解 JDK 與 JRE 安裝目錄下的東西之後，這邊作個總結，您到底要記得哪些東西？答案是 JDK 安裝目錄下的「bin」目錄，因為當您撰寫完 Java 程式之後，無論是編譯或執行程式，都會使用到「bin」目錄下所提供的工具程式。

2.2 設定 Path 與 Classpath

對於習慣圖形化介面操作的初學者而言，在文字模式下執行程式是一件陌生的事，也因此不瞭解 Path 路徑設定的方法與作用，而 Java 執行的平台也有自己的一套路徑規則來找尋撰寫好的 Java 類別，也就是所謂的 Classpath 設定，這個小節中將告訴您如何進行這些相關的設定。

2.2.1 設定 Path

在安裝好 JDK 程式之後，在JDK安裝目錄（假設是 C:\Program Files\Java\jdk1.6.0）下的「bin」目錄，會提供一些開發 Java 程式中必備的工具程式，對於 Java 的初學者我所給的建議是從文字模式（在 Windows 2000/XP 下稱之為命令提示字元）下來操作這些工具程式，您可以在 Windows 2000/XP 的「開始」選單中選擇「執行」，鍵入「cmd」指令來開啟文字模式。

雖然您知道 JDK 的工具程式是位於「bin」目錄下，但您的作業系統並不知道如何找到這些工具程式，所以當您鍵入「javac」嘗試執行編譯程式時，文字模式下會告訴您找不到 javac 工具程式。

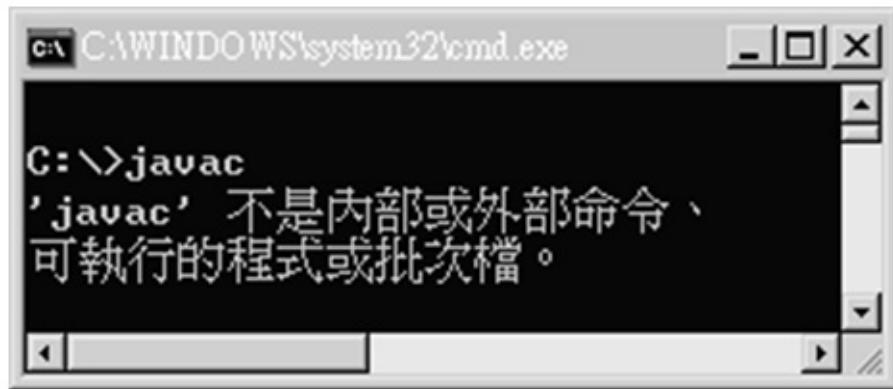


圖 2.8 出現這個訊息表示找不到指定的工具程式

您必須告訴作業系統，應該到哪些目錄下嘗試找到您所想使用的工具程式，有幾個方法可以進行這項設定，最方便的方法就是設定系統變數中的 Path 環境變數，在 Windows 2000/XP 下可以選擇桌面上的「我的電腦」並按滑鼠右鍵，選擇「內容」後切換至「進階」頁面，並按下方的「環境變數」按鈕，在「環境變數」對話方塊中編輯「Path」變數。



圖 2.9 選擇 Path 變數並按下「編輯」以進行路徑設定

在 Path 變數的「編輯系統變數」的對話方塊中，於「變數值」欄位的最前方輸入 JDK 「bin」 目錄的路徑（這邊假設是 C:\Program Files\Java\jdk1.6.0\bin），然後緊跟著一個分號，以作為路徑設定的區隔，接著按「確定」即可完成設定。



圖 2.10 在 Path 變數中加入 JDK 的 bin 目錄路徑

之所以要將 JDK 的路徑設定放置在 Path 變數設定的最前方，是因為作業系統在搜尋 Path 路徑設定時，會從最前方開始讀取，如果在路徑下找到指定的程式，就會直接執行，當您的系統中安裝有兩個以上的 JDK 時，在 Path 路徑中設定的順序，將決定執行哪個 JDK 下的工具程式。

設定 Path 變數之後，您要重新開啟一個文字模式才能重新讀入 Path 變數內容，接著如果您執行 javac 程式，您應該可以看到圖 2.11 的畫面。

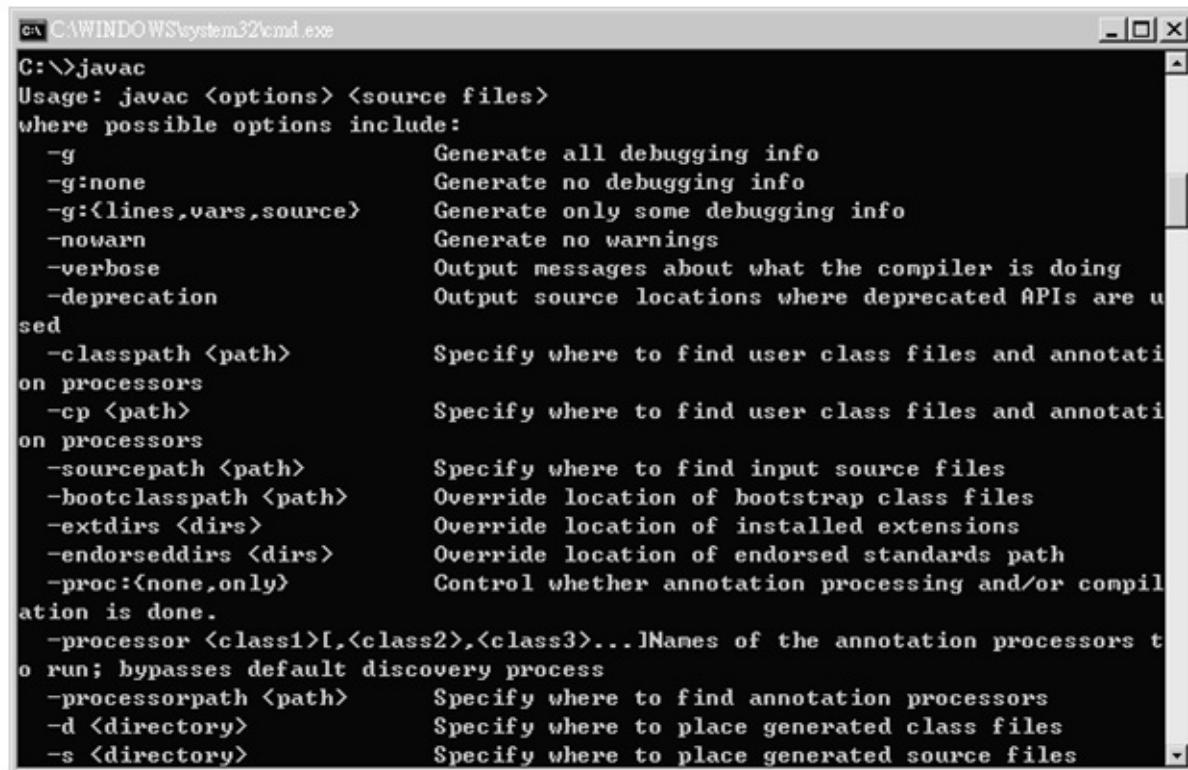


圖 2.11 設定 Path 變數成功的話，就可以找到指定的工具程式

您也可以在文字模式下執行以下的指令，以直接設定目前的環境變數包括 Path 變數（這個設定在下次重新開啟文字模式時就不再有效）：

```
set Path= C:\Program Files\Java\jdk1.6.0\bin;%Path%
```

事實上，如果您沒有在 Path 變數設定 JDK 的 bin 路徑的話，也可以直接執行 java 工具程式，這是因為 Windows 下安裝 JRE 時，會將 java.exe 複製至「C:\Windows\System32」路徑之下，而這個路徑在 Path 變數中是預設的路徑。

如果 Path 設定是尋找 JDK 安裝目錄下的「bin」目錄，則作業系統會找尋 JDK 安裝目錄下工具程式，因而當您執行 java 程式時，就會使用 JDK 所附的 JRE（即「C:\Program Files\Java\jdk1.6.0\jre」），而不是公用的 JRE（即 C:\Program Files\Java\jre1.6.0\）。

簡單的說，作業系統會嘗試在您指定的 Path 變數中尋找指定的工具程式，當您鍵入 javac 指令時，由於 Path 變數中有設定 JDK 的「bin」目錄之路徑，作業系統就可以根據這個訊息來找到 javac、java 等工具程式。

2.2.2 設定 Classpath

Java 執行環境本身就是一個平台，執行於這個平台上的程式是已編譯完成的 Java 程式（之後會介紹到 Java 程式編譯完成之後，會以 .class 檔案存在），如果將 Java 執行環境比喻為作業系統的話，如果設定 Path 變數是為了讓作業系統找到指定的工具程式（以 Windows 來說的話就是找到 .exe 檔案），則設定 Classpath 的目的就是為了讓 Java 執行環境找到指定的 Java 程式（也就是.class 檔案）。

有幾個方法可以設定 Classpath，最簡單的方法是在系統變數中新增 Classpath 環境變數，在圖 2.9 中的「系統變數」按下「新增」鈕，在「變數名稱」欄位中輸入「Classpath」，在「變數值」欄位中輸入 Java 類別檔案的位置，例如可以輸入「.;C:\Program Files\Java\jdk1.6.0\lib\tools.jar; C:\Program Files\Java\jre1.6.0\lib\rt.jar」（jar 檔是 zip 壓縮格式，當中就包括了 .class 檔案以及 jar 中的 Classpath 設定），每一筆資料中間必須以「;」作為分隔。

事實上 JDK 6 預設就會到現行工作目錄（上面的「.' 設定），以及 JDK 的「lib」目錄（這邊假設是 C:\Program

Files\Java\jdk1.6.0\lib) 中尋找 Java 程式，所以如果您的 Java 程式是在這兩個目錄中，則不必設定 Classpath 變數也可以找的到，將來如果您的 Java 程式不是放置在這兩個目錄時，則可以如上設定 Classpath。

如果所使用的 JDK 工具程式具有 Classpath 指令選項，則可以在執行工具程式時一併指定 Classpath，例如：

```
javac -classpath classpath1;classpath2 ...
```

其中 classpath1、classpath2 是您實際要指定的路徑；您也可以在文字模式下執行以下的指令，以直接設定目前的環境變數包括 Classpath 變數（這個設定在下次重新開啟文字模式時就不再有效）：

```
set CLASSPATH=%CLASSPATH%;classpath1;classpath2...
```

總而言之，設定 Classpath 的目的，在於告訴 Java 執行環境，哪些目錄下可以找到您所要執行的 Java 程式。一個分辨 Path 與 Classpath 的方式就是：「對於 Windows 作業系統來說，Path 是讓作業系統可以找到 ".exe" 執行檔的存在，而對於 Java 執行環境來說，ClassPath 就是讓 JVM 可以找到 ".class" 執行檔的存在」。

良葛格的話匣子 在 Design Patterns Elements of Reusable Object-Oriented Software 書中對「框架」作出的解釋是：
框架就是一組互相合作的類別組成，它們為特定類型的軟體開發提供了一個可以重複使用的設計。

2.3 第一個 Java 程式

完成 JDK 相關環境設定之後，無論如何就先寫個簡單的 Java 程式，以測試一下環境設定是否正確，順便增強一些學習的信心，以下要介紹的第一個 Java 程式是會顯示 "嗨！我的第一個 Java 程式！" 這段訊息的簡單程式。

2.3.1 撰寫、編譯 Java 程式

在正式撰寫程式之前，請先確定您可以看的到檔案的副檔名，在 Windows 2000/XP 下預設是不顯示檔案的副檔名，這會造成您重新命名檔案時的困擾，如果您目前在「檔案總管」下無法看到檔案的副檔名，請先執行工具列上的「工具/資料夾選項」並切換至「檢視」頁面，取消「隱藏已知檔案類型的副檔名」之選取。

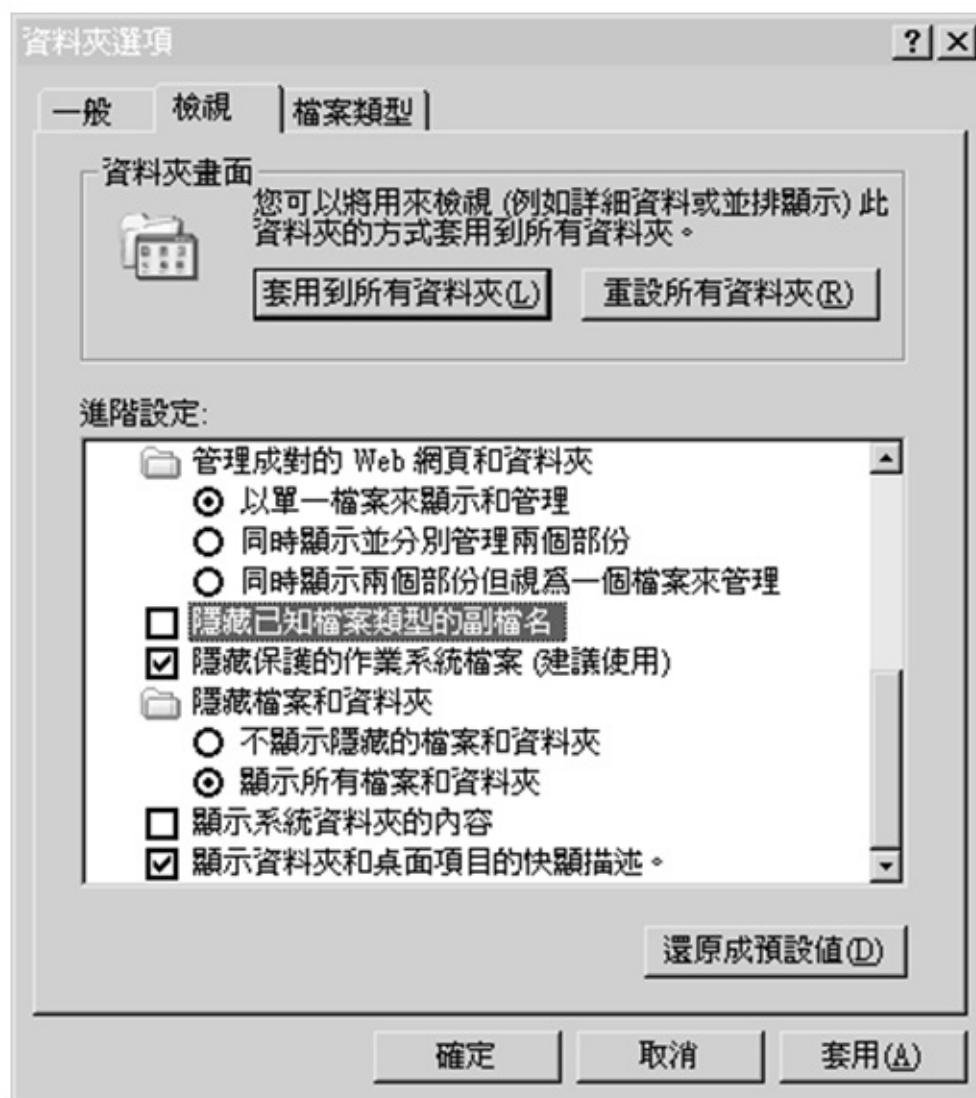


圖 2.12 取消「隱藏已知檔案類型的副檔名」以顯示副檔名

接著選擇一個目錄來撰寫 Java 原始碼檔案，假設是檔案會儲存在 C:\workspace 目錄，您要新增一個「文字文件」（也就是 .txt 文件），並重新命名文件為「HelloJava.java」，由於您是將文字文件的副檔名從 .txt 改為 .java，所以系統會詢問您是否更改副檔名，請確定更改，接著在「HelloJava.java」上按兩下開啟檔案，並照著圖 2.13 撰寫程式。



圖 2.13 第一個 Java 程式

這個程式有幾點必須注意：

- Java 的原始檔案必須以副檔名 .java 作結束

這也就是您必須讓「檔案總管」顯示副檔名的原因。

- 主檔名與類別名稱必須一致

Java 程式的類別名稱是指 "class" 關鍵字 (Keyword) 後的名稱，就這個例子而言，也就是 HelloJava 這個名稱，這個名稱必須與 HelloJava.java 的主檔名 (HelloJava) 一致。

- 注意每個字母的大小寫

Java 程式中會區分字母的大小寫，System 與 system 對 Java 程式來說是不一樣的名稱。

- 空白只能是半型空白字元或是Tab字元

有些初學者可能不小心輸入了全型空白字元，這很不容易檢查出來。

撰寫好程式並存檔後，接著開啟一個文字模式，並使用以下的指令切換至 HelloJava.java 所在的目錄（在此例中是 C:\workspace）：

```
> cd C:\workspace
```

接著使用 javac 工具程式來編譯 .java 檔案，這邊指定編譯 HelloJava.java，指令如下：

```
C:\workspace\javac HelloJava.java
```

如果編譯順利成功，則在 HelloJava.java 所在的目錄，會出現一個 HelloJava.class 的檔案，這是已經編譯完成的 Java 位元碼（bytecodes）檔案程式。

您可能會遇到以下的幾個錯誤，這表示您有一些地方操作有誤：

- error: cannot read: HelloJava.java

這表示 javac 工具程式找不到您指定的 .java 檔案，請檢查檔案是否存在目前的目錄中，或是檔案名稱是否有誤。

- HelloJava.java:1: class HelloJava is public, should be declared in a file named HellJava.java

類別名稱與主檔名不符，請確定主檔名與類別名稱是相同的。

- HelloJava.java:3: cannot find symbol

程式碼中某些部份打錯了，最常發生的原因可能是沒有注意到字母大小寫。

- 'javac' 不是內部或外部命令、可執行的程式或批次檔。

Path 設定有誤或沒有在 Path 中加入 JDK 的「bin」目錄，請參考前一節的內容。

2.3.2 執行 Java 程式

在順利編譯出 .class 的檔案之後，可以使用 java 工具程式來執行它，執行時必須指定類別名稱，就以上的例子來說，也就是指定 HelloJava.class 這個檔案的主檔名，指令執行方式如下：

```
C:\workspace>java HelloJava
```

java 工具程式會根據您指定的類別名稱，實際載入 .class 的檔案，以上例來說，就是載入 HelloJava.class 並執行，執行的結果應該是圖 2.14 的畫面。



圖 2.14 執行第一個 Java 程式的成功畫面

您可能會遇到以下的幾個錯誤，這表示您有一些地方操作有誤：

- Exception in thread "main" java.lang.NoClassDefFoundError

java 工具程式找不到您所指定的類別，請確定指定的類別存在目前的目錄中，名稱是否指定正確（如果是舊版的 JDK，可能必須在 Classpath 中加入 ". "，表示可於目前目錄中找到所指定的類別）。

- Exception in thread "main" java.lang.NoSuchMethodError: main

沒有指定 Java 程式的進入點（Entry point），java 工具程式指定的類別必須要有一個程式進入點，也就是必須包括 main(String[] args) 這個方法（method），請確定您的程式中包括它，必須與圖 2.13 的 HelloJava.java 內容一致。

在之後的章節，如果不是要特別強調的話，不再重複指出如何編譯與執行 Java 程式，在往後的章節中說要編譯 Java 程式時，就是指使用 `javac` 工具程式來編譯 `.java`，而說要執行 Java 程式時，就是指使用 `java` 工具程式來執行指定的 Java 類別。

2.4 選擇開發工具

從學習的角度來說，建議初學者使用純文字檔案來撰寫 Java 程式，並在文字模式下親自使用工具程式來編譯、執行 Java 程式，藉此來瞭解關於 Path、Classpath，熟悉工具程式的使用，習慣一些撰寫 Java 程式所必須注意的地方，嘗試從文字模式所提供的訊息中瞭解所撰寫的程式發生什麼問題，以及如何改正這些問題。

當然只使用純文字檔案總是有相當多的不便，在學習 Java 一段時間之後，您也許會想要找尋一個好用的 Java 開發工具，這邊的建議是先從簡單的文字編輯輔助工具開始，像是 [UltraEdit](#) 或 [Editplus](#)，這兩個文字編輯輔助工具都有語法標示顯示，以及一些好的尋找、取代、比較等功能，在Linux下的話，我經常使用 vi 來編輯 Java 程式。

從開發效率的角度來看，選擇一個好的整合開發環境（Integrated Development Environment, IDE）是必要，使用何種開發整合開發環境，依開發團隊的需求而各有不同，如果沒有什麼特別的需求，建議可以從簡單的 [JCreator](#) 或 [BlueJ](#) 開始，這些開發環境不僅提供語法顯示等方便程式編輯的功能，也整合了一些 Java 工具程式的使用，使用上並不會太複雜，作為在文字編輯程式之後的進階開發環境程式，是個不錯的嘗試。

如果想要使用功能更齊全的 IDE，[JBuilder](#)、[Eclipse](#)、[NetBeans](#) 等都是功能齊全的 IDE，也都各有其支持者，但如果想找一個可以免費使用而功能齊全的 IDE 的話，建議可以使用 Eclipse 或 NetBeans。

Eclipse 是個可以免費下載、使用的IDE，但並不因為它免費而缺少必要的功能，Eclipse 提供了 JDT（Java Development Tools），這是基於 Eclipse 且功能齊全的 Java IDE（Eclipse 並不只被用來開發 Java 程式），它所提供的 PDE（Plug-In Development Environment）可以讓您對 JDT 進行功能擴展，在網路上總是可以找到 Plug-In 程式來加入 Eclipse 中以增加您所想要的功能，必要時，您也可以開發自己的 Plug-In 來為 Eclipse 擴展功能，Eclipse 也是大多數的 Java 開發人員所推薦的 IDE。

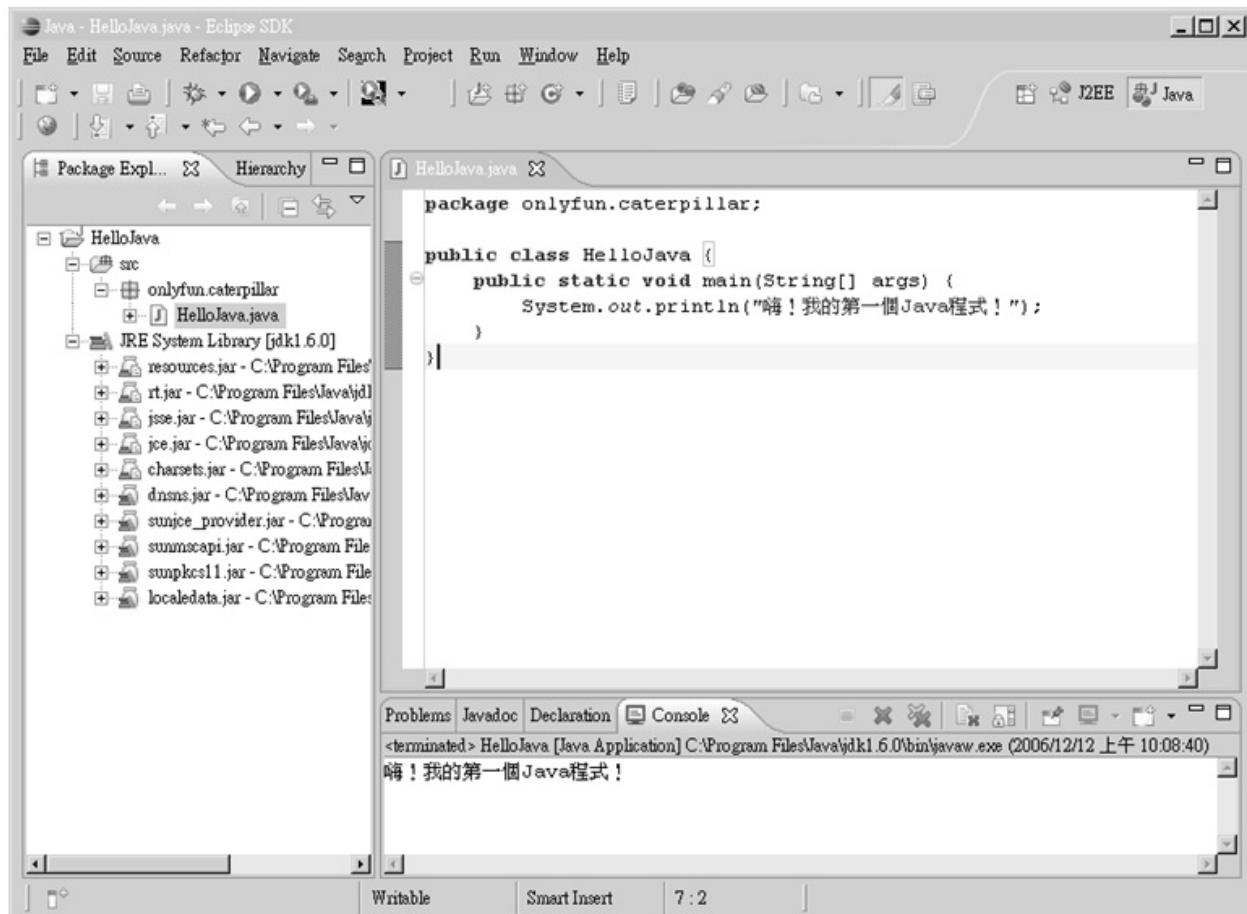


圖 2.15 Eclipse 是個功能強大且 free 的 IDE

NetBeans IDE 是 Sun 官方所推薦下載的 IDE，在 Java 官方網站上也可以下載綁定 NetBeans IDE 的 JDK 安裝檔案，NetBeans 本身是個高度模組化的 IDE，可以讓您下載新功能模組來擴充 IDE 的功能。在 NetBeans 在 5.0 版本之後，功能已相當成熟，每一次新的 JDK 版本釋出之後，NetBeans IDE 總是最先支援新版 JDK 的 IDE，有相當多的 Java 開發人員也推薦使用 NetBeans IDE。

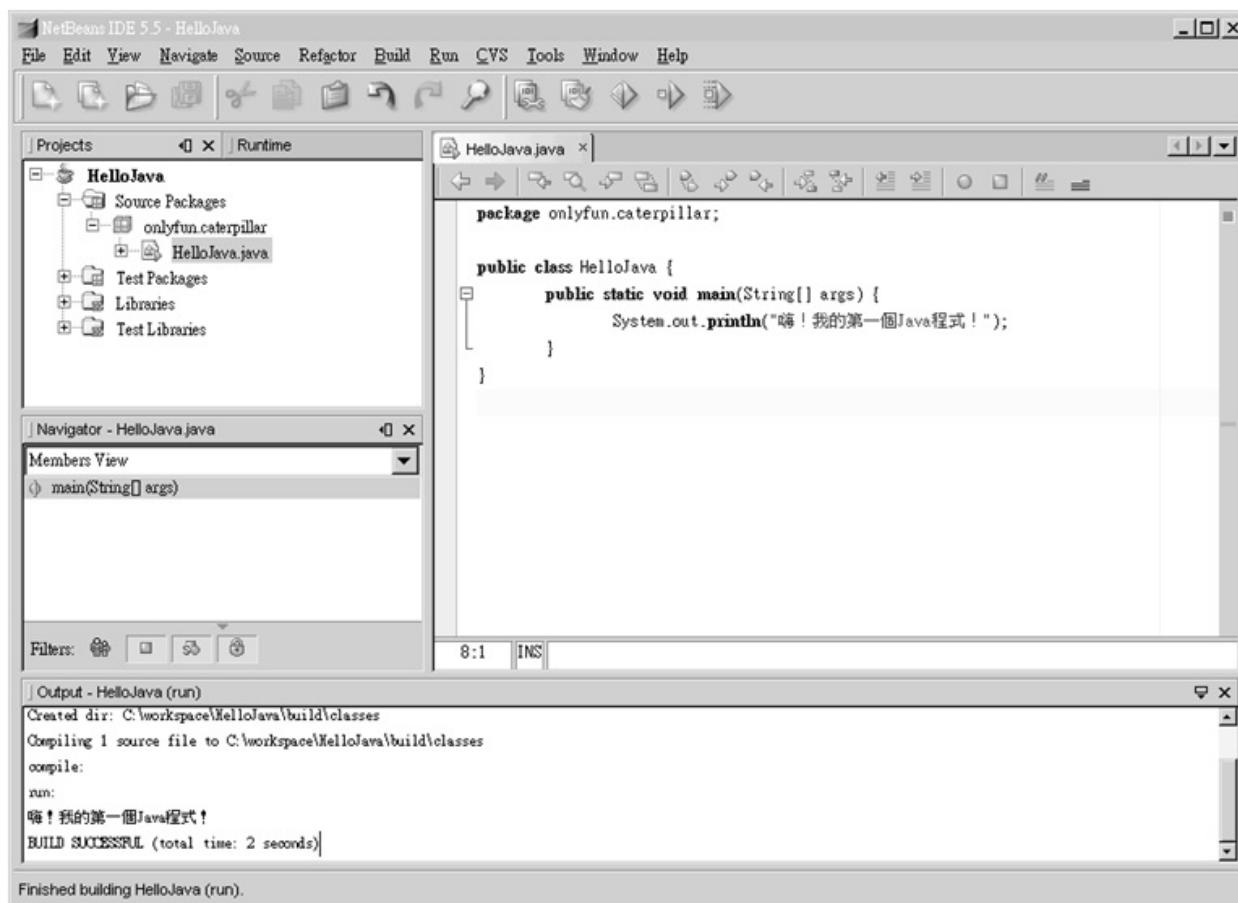


圖 2.16 NetBeans 是 Sun 官方所推薦的 IDE

良葛格的話匣子 在我學習 Java 的過程中，差不多有兩年的時間，都是使用純文字檔案撰寫 Java 程式，這使得我對 Java 能有深入的瞭解，因而我建議初學者（完全沒有程式經驗的使用者）在學習 Java 的過程中，也從純文字檔案撰寫 Java 中開始學習，在您真正需要開發程式（或團隊需求），或想學習一個 IDE 的使用時，才開始使用 IDE 開發 Java 程式，不要只是偷懶或貪圖方便而使用 IDE。

2.5 接下來的主題

每一個章節的內容由淺至深，初學者該掌握的深度要到哪呢？在這個章節中，對於初學者我建議至少掌握以下幾點內容：

- 能自行下載、安裝 JDK
- 瞭解 JDK、JRE 的安裝位置
- 懂得設定 Path 之目的
- 懂得設定 Classpath 之目的
- 能成功的自行撰寫並執行第一個 Java 程式

之後的章節將以這一個章節的內容作為基礎進行說明，首先將從 Java 的資料型態、變數常數、流程控制開始介紹，這也是學習任何一個程式語言都不可缺少的過程。

第 3 章 語法入門

程式語言（Programming Language）本質上就是一個語言，語言的目的在於讓您與特定對象進行溝通，只不過程式語言的溝通對象是電腦。現在您學習的是 Java 語言，用 Java 寫程式的目的就是在告訴電腦，您希望它為您作哪些事，Java 既然是語言，就有其規定的語法，這個章節就是在學習基本的 Java 語法。

我建議初學者從文字模式學習如何操作程式，您在這個章節中將會學到：如何告訴電腦在命令模式下顯示訊息？什麼時候該等待您輸入資料（即 Console 互動）？程式中您要使用的資料有哪些（即資料型態）？您要用這些資料進行什麼樣的運算（即各種運算子的使用）？在某些情況滿足時下一步要作什麼？不滿足某些條件時又該作些什麼（即流程控制）？

3.1 第一個 Java 程式

先來解釋一下第 2 章中您所完成的「第一個 Java 程式」，如果您學過 C 語言，對於 3.1.2 「給 C 使用者的第一個 Java 程式」可能會覺得比較親切，因為介紹了類似 C 語言中 printf() 函式的功能，另外別忘了在寫程式時加入一些註解文字，這在閱讀程式碼的時候會很有幫助。

3.1.1 解釋第一個 Java 程式

要對新手解釋第一個 Java 程式事實上並不簡單，因為一個最簡單的 Java 程式就會涉及檔案管理、類別（Class）、主程式、命令列引數（Command line argument）等觀念，我很想對您說，反正一個基本的 Java 程式就這麼寫就對了，因為要一下子接受這麼多觀念並不容易。總之，如果現階段您無法瞭解，就請當它是 Java 語言的文法規範。

先重新列出您的第一個 Java 程式以方便說明。

範例 3.1 HelloJava.java

```
public class HelloJava {
    public static void main(String[] args) {
        System.out.println("嗨！我的第一個Java程式！");
    }
}
```

- 定義類別（Class）

撰寫 Java 程式通常都是由定義「類別」開始，"class" 是 Java 用來定義類別的關鍵字，範例中類別的名稱是 HelloJava，這與您編輯的檔案（HelloJava.java）主檔名必須相同，HelloJava 類別使用關鍵字 "public" 告知，在編寫 Java 程式時，一個檔案中可撰寫數個類別，但是只能有一個公開（public）類別，而且檔案主檔名必須與這個公開類別的名稱相同，在定義類別名稱時，建議將類別首字母大寫，並在類別名稱上表明類別的作用。

- 定義區塊（Block）

在 Java 程式使用大括號 '{' 與 '}' 來定義區塊，大括號兩兩成對，目的在區別定義的作用範圍，例如您的程式中，HelloJava 類別的區塊包括了 main()，而 main() 的區塊包括了一句顯示訊息的程式碼。

- 定義 main() 方法（Method）

main() 是 Java 程式的「進入點」（Entry point），程式的執行是由進入點開始的，類別中的方法是類別的成員（Member），main() 方法一定是個 "public" 成員（member），這樣它才可以執行環境被呼叫；main() 方法不需要產生物件（Object）就能被執行，所以它必須是個 "static" 成員，"public" 與 "static" 的觀念都是 Java 物件導向（Object-oriented）程式的觀念，之後專門討論類別與物件時會再介紹，這邊先不用理解 "public" 與 "static" 的真正意涵。

main() 之前的「void」表示 main() 執行結束後不傳回任何值，Java 程式的 main() 方法不需傳回任何值，所以一律宣告 void；main() 括號間的 String[] args 可以在執行程式時取得使用者指定的命令列引數（Command line argument），目前雖然您用不到，但仍要撰寫它，這是規定。

- 撰寫陳述（Statement）

來看 main() 當中的一行陳述：

```
System.out.println("嗨！我的第一個Java程式！");
```

陳述是程式語言中的一行指令，簡單的話就是程式語言的「一句話」。注意每一句陳述的結束要用分號 ';'，在上面的第一個 Java 程式

陳述中，您使用了 `java.lang` 套件（package）下的 `System` 類別的公開（public）成員 `out` 物件，`out` 是一個 `PrintStream` 物件，您使用了 `PrintStream` 所提供的 `println()` 方法，將當中指定的字串（String）"嗨！我的第一個Java 程式！" 輸出至文字模式上。

注意在 Java 中字串要使用 "" 包括，`println()` 表示輸出字串後自動換行，如果使用 `print()` 的話，則輸出字串後程式並不會自動斷行。

一個最基本的 Java 程式完成了，接下來要編譯程式了，關於編譯在第 2 章中有介紹過，這邊再作個提示，編譯時是使用 `javac` 公用程式，如下所示：

```
javac HelloJava.java
```

編譯完成後，預設在同一個目錄下會產生一個 `HelloJava.class` 的位元碼（bytecodes）檔案，在執行時期可以由執行環境轉換為平台可執行的機器碼，執行程式時是使用 `java` 工具程式，如下所示：

```
java HelloJava
```

注意最後並沒有加上 `*.class` 的副檔名，您只要提供類別名稱就可以了，程式畫面會顯示如下：

```
嗨！我的第一個Java程式！
```

良葛格的話匣子 定義區塊的風格因人而異，有的設計人員習慣先換行再定義區塊，例如：

```
public class HelloJava
{
    public static void main(String[] args)
    {
        System.out.println("嗨！我的第一個Java程式！");
    }
}
```

這麼作的好處是可以很快的找到兩兩成對的大括號，區塊對應清楚。找一個您喜歡的風格撰寫，以清晰易讀為原則就可以了。

3.1.2 紿 C 使用者的第一個 Java 程式

學過 C 語言的設計人員總是對 `printf()` 方法的功能總是難以忘懷，他們在學 Java 語言時常不免感慨：「是的！是 `printf()`，他們忘了把 `printf()` 放進去了....」。

在某些時候，`printf()` 方法中字串上可以指定引數來進行輸出的功能確實令人難以割捨，幸好，Java 在 J2SE 5.0 這個版本中，總算給了 C 使用者類似 `printf()` 的功能了，如果您是學過 C 的使用者，下面這第一個 Java 程式或許會讓您高興一些：

範例 3.2 HelloJavaForC.java

```
public class HelloJavaForC {
    public static void main(String[] args) {
        System.out.printf("%s！這是您的第一個Java程式！\n",
                          "C語言Fan");
    }
}
```

這次使用的是 `out` 物件的 `printf()` 方法，'`%s`' 對應於第一個字串 "C語言Fan"，程式的輸出會是如下：

C語言Fan！這是您的第一個Java程式！

'\n' 對 C 程式設計人員並不陌生，它是換行字元，您也可以如下使用 `println()` 進行換行。

範例 3.3 SecondJavaForC.java

```
public class SecondJavaForC {
    public static void main(String[] args) {
        System.out.printf("%s！這是您的第二個Java程式！",
                          "C語言Fan").println();
    }
}
```

程式的輸出會是如下：

C語言Fan！這是您的第二個Java程式！

在 `printf()` 要指定數字對應的話，可以使用 '%d'，例如：

範例 3.4 ThirdJavaForC.java

```
public class ThirdJavaForC {
    public static void main(String[] args) {
        System.out.printf("%s！這是您的第 %d 個Java程式！\n",
                          "C語言Fan", 3);
    }
}
```

'%s' 對應至 "C語言Fan"，而 '%d' 對應至數字 3，所以程式的輸出會是如下：

C語言Fan！這是您的第 3 個Java程式！

這邊再作一次提醒，`printf()` 方法是在 J2SE 5.0 之後才有的功能，1.4 或更早版本的 JDK 並沒有辦法編譯範例 3.2、3.3 與 3.4。

良葛格的話匣子 如果您安裝的是 JDK6，即使您沒有使用到 JDK6 的新功能，所編譯出來的 class 檔預設也是無法在更早版本的 JRE 上運行的，因為 JDK6 編譯出來的 class 檔版號跟較早版本的 JVM 所接受的版號並不相同。

在不使用 JDK6 的新功能下，如果您希望編譯出來的 class 可以在 1.5 版本或之前版本上的環境上運行，則編譯時必須指定 `-source` 與 `-target` 引數，`-source` 指定原始碼中可以使用的版本功能，`-target` 指定編譯出來的位元碼適用的虛擬機器版本，例如您想要編譯出來的檔案在 1.5 版本環境上運行的話，可以如下編譯程式：

```
javac -source 1.5 -target 1.5 HelloJava.java
```

3.1.3 為程式加入註解

在撰寫程式的同時，您可以為程式碼加上一些註解（Comment），說明或記錄您的程式中一些要注意的事項，Java 語言是您用來與電腦溝通的語言，而註解的作用則是與開發人員溝通。

原始碼檔案中被標註為註解的文字，編譯器不會去處理它，所以註解文字中撰寫任何的東西，對編譯出來的程式不會有任何影響。在 Java 中可以用兩種方式為程式加上註解，以範例 3.4 為例，您可以為它加上一些註解文字，例如：

```
/*
 * 作者：良葛格
 * 功能：示範printf()方法
 * 日期：2005/4/30
 */
public class ThirdJavaForC {
    public static void main(String[] args) {
        // printf() 是J2SE 5.0的新功能，必須安裝JDK 5.0才能編譯
        System.out.printf("%s！這是您的第 %d 個Java程式！\n",
                           "C語言Fan", 3);
    }
}
```

'/' 與 '/' 用來包括跨行的註解文字，通常開發人員為了讓註解文字看來比較整齊清晰，中間還會使用一些 '' 來排版，只要記得使用多行註解時，是以 '/' 開始註解文字，以 '*' 結束註解文字，所以您不能用巢狀方式來撰寫多行註解，例如下面的方式是不對的：

```
/*
 * 註解文字1.....bla..bla
 *
 *     註解文字2.....bla..bla
 */
*/
```

編譯器在處理前面的撰寫方式時，會以為倒數第二個 '/' 就是註解結束的時候，因而對最後一個 '/' 就會認為是錯誤的符號，這時就會出現編譯錯誤的訊息。

'//' 則可以用來撰寫單行註解，在 '//' 之後的該行文字都被視為註解文字，多行註解可以包括單行註解，例如：

```
/*
 * 註解文字1.....bla..bla
 *     // 註解文字2.....bla..bla
 */
```

註解的撰寫時機與內容並沒什麼特別的規定，以清晰易懂為主，註解的目的在於說明程式，是給開發人員看的，您可以使用註解在程式中寫下重要事項，日後作為備忘或是方便其他開發人員瞭解您的程式內容。

註解的另一個作用則是用來暫時註銷某些陳述句，當您覺得程式中某些陳述有問題，可以使用註解標示起來，如此編譯器就不會去處理，例如同樣以範例 3.4 為例：

```
public class ThirdJavaForC {
    public static void main(String[] args) {
        // System.out.println("C語言Fan！這是您的第3個Java程式！");
        System.out.printf("%s！這是您的第 %d 個Java程式！\n",
                           "C語言Fan", 3);
    }
}
```

在'//'之後的內容被視為註解文字，編譯器不會去處理，所以執行時不會執行該行陳述，如果日後要重新恢復那些陳述句的功能，只要將註解符號消去就可以了。# 第一個 Java 程式

3.2 在文字模式下與程式互動

從互動中學習，是我最喜愛的學習方式，我建議學習 Java 的第一步，要先能看到您的程式執行結果，要可以對程式輸入一些資料，作一些傻瓜式的互動。

3.2.1 使用 Scanner 取得輸入

在文字模式下要輸入資料至程式中時，您可以使用標準輸入串流物件 System.in，然而實際上很少直接使用它，因為 System.in 物件所提供的 read() 方法，是從輸入串流取得一個位元組的資料，並傳回該位元組的整數值，但通常您要取得的使用者輸入會是一個字串，或是一組數字，所以 System.in 物件的 read() 方法一次只讀入一個位元組資料的方式並不適用。

在 J2SE 5.0 中，您可以使用 java.util.Scanner 類別取得使用者的輸入，java.util 指的是套件（package）層級，java.util.Scanner 表示 Scanner 這個類別是位於 java/util 這樣的階層之下，現階段您可以將這個階層想像為類似檔案管理的目錄階層。直接使用實例來看看如何使用 Scanner 取得使用者的輸入字串：

範例 3.5 ScannerDemo.java

```
import java.util.Scanner;

public class ScannerDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("請輸入您的名字：");
        System.out.printf("哈囉！ %s!\n", scanner.next());
    }
}
```

先看看執行結果再來解釋程式的內容：

```
請輸入您的名字：良葛格
哈囉！ 良葛格！
```

java.util 套件是 Java SE 的標準套件，您使用 import 是在告訴編譯器，您將使用 java.util 下的 Scanner 類別。

new 關鍵字表示您要新增一個 Scanner 物件，在新增一個 Scanner 物件時需要一個 System.in 物件，因為實際上還是 System.in 在取得使用者的輸入，您可以將 Scanner 看作是 System.in 物件的支援者，System.in 取得使用者輸入之後，交給 Scanner 作一些處理（實際上，這是 Decorator 模式的一個應用，請見這一章後的索引）。

不以 Java 術語而以白話來說，您告訴執行環境給您一個叫作 Scanner 的工具，然後您可以使用這個工具的 next() 功能，來取得使用者的輸入字串，但要如何取得數字呢？您可以使用 Scanner 工具的 nextInt() 功能，例如：

範例 3.6 ScannerDemo2.java

```
import java.util.Scanner;

public class ScannerDemo2 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("請輸入一個數字： ");
        System.out.printf("您輸入了 %d !\n",
                         scanner.nextInt());
    }
}
```

`nextInt()` 會將試著取得的字串轉換為 int 型態的整數，關於資料型態（Data type）我待會就會介紹，先來看看執行結果：

```
請輸入一個數字： 100
您輸入了 100 !
```

依同樣的方式，您還可以使用 Scanner 的 `nextFloat()`、`nextBoolean()` 等方法來取得使用者的輸入，並轉換為正確的資料型態。

在 JDK6 之中，您還可以使用 `System.console()` 方法取得一個 `java.io.Console` 物件，利用它來取得使用者輸入非常的方便，關於 Console 的介紹，請見第 21 章關於 JDK6 新功能的介紹。

3.2.2 使用 BufferedReader 取得輸入

Scanner 取得輸入的依據是空白字元，舉凡按下空白鍵、tab 鍵或是 enter 鍵，Scanner 就會傳回下一個輸入，所以在某些時候並不適用，因為使用者可能輸入一個字串，中間會包括空白字元，而您希望取得完整的字串，如果您想要取得包括空白字元的輸入，比較簡單的方法是使用 `java.io.BufferedReader` 類別取得輸入，這個方法也是在 Java SE 1.4 或之前版本下取得使用者輸入的方式。

必須先說明的是，關於使用 BufferedReader 來取得輸入，在理解上要複雜的多，我在這邊先加以說明，但如果您的現階段無法理解的話不用氣餒，因為使用方法是固定的，每次使用前先如法泡製就可以了，在第 14 章介紹完輸入輸出，您就會理解 BufferedReader 的運作方式。

BufferedReader 類別是 `java.io` 套件中所提供之一個類別，所以使用這個類別時必須使用 `import` 告訴編譯器這個類別位於 `java.io` 套件下。

使用 BufferedReader 物件的 `readLine()` 方法必須處理 `java.io.IOException` 例外（exception），例外處理機制是 Java 提供給程式設計人員捕捉程式中可能發生的錯誤所提供的機制，現階段您處理 `IOException` 的方法是在 `main()` 方法加上 `throws IOException`，這在第 10 章介紹例外處理時，我會再詳細介紹為何要這麼作。

BufferedReader 在建構時接受一個 `java.io.Reader` 物件，在讀取標準輸入串流時，您可以使用 `java.io.InputStreamReader`，它繼承（Inherit）了 Reader 類別，您可以使用以下的方法來為標準輸入串流建立緩衝區物件：

```
BufferedReader bufferedReader = new BufferedReader(
    new InputStreamReader(System.in));
```

BufferedReader bufferedReader 表示宣告一個型態為 BufferedReader 的參考名稱，而 `new BufferedReader()` 表示您以 BufferedReader 類別建構一個物件，`new InputStreamReader(System.in)` 表示接受一個 System.in 物件來建構一個 InputStreamReader 物件。

不用 Java 藝語而用白話來解釋上一段的話，就是您增加一個 BufferedReader 工具，這個工具中還要加上一個 InputStreamReader 工具，而 InputStreamReader 工具中實際的核心是 System.in 工具，這三個工具組合在一起，就可以讓您進行文字輸入的讀取。

如果還是不理解的話沒關係，照著抄寫就可以了，實際看個例子瞭解如何使用 BufferedReader 來讀取輸入。

範例 3.7 BufferedReaderDemo.java

```
import java.io.*;

public class BufferedReaderDemo {
    public static void main(String[] args) throws IOException {
        BufferedReader bufferedReader =
            new BufferedReader(
                new InputStreamReader(System.in));
```

```

        System.out.print("請輸入一列文字，可包括空白：");
        String text = bufferedReader.readLine();
        System.out.println("您輸入的文字：" + text);
    }
}

```

由於這次您所使用到的 BufferedReader、InputStreamReader 與 IOException 等類別，都是位在 java.io 套件下，所以在程式的一開頭可以使用 import 與 * 號，告訴編譯器到 java.io 下找這些類別。

readLine() 方法會傳回使用者在按下 Enter 鍵之前的所有字元輸入，不包括最後按下的 Enter 返回字元，程式的執行範例如下所示：

```

請輸入一列文字，可包括空白：Have a nice day :)
您輸入的文字：Have a nice day :)

```

良葛格的話匣子 學習一個新的事物時，如果遇到一些觀念無法馬上理解，這可能是因為要理解觀念會需要其它觀念先建立起來，所以先暫時放下這個疑問也是一個學習方法，稱之為「闕疑」，在往後的學習過程中待必要的觀念學會後，目前的疑問自然也會解開。

但什麼該闕疑？什麼觀念又該先徹底理解？在這本書中我會根據經驗告訴您，在告訴您闕疑的同時，也會告訴您這些觀念實際上在哪個章節（或網路資料上）會詳細說明。

3.2.3 標準輸入輸出串流

在之前的範例程式中，您使用了 System 類別中的靜態物件 out，它提供標準輸出串流（Stream）輸出，會在程式開始執行之後自動開啟並準備接受指定的資料，它通常對應至顯示輸出（文字模式、終端機輸出），您可以將輸出重新導向至一個檔案，只要執行程式時使用 '>' 將輸出結果導向至指定的檔案，例如：

```
java HelloJava > HelloJavaResult.txt
```

上面的執行會將原本文字模式下的顯示結果導向至 HelloJavaResult.txt，而不會在螢幕上顯示訊息，HelloJavaResult.txt 將會有執行的結果文字“嗨！我的第一個Java程式！”。

除了標準輸出串流 out 之外，Java 程式在執行之後，還會開啟標準輸入串流 in 與標準錯誤輸出串流 err。標準輸入串流 in 也是 System 類別所提供的靜態物件，在程式開始之後它會自動開啟，對應至鍵盤或其它的輸入來源，準備接受使用者或其它來源的輸入，您可以使用 read() 方法來讀取輸入，不過通常很少直接使用，而會使用一個 Scanner 物件或 BufferedReader 來讀取輸入，方法我已經在前面的內容介紹過了。

標準錯誤輸出串流 err 也是在程式執行後自動開啟，它會將指定的字串輸出至顯示裝置或其它指定的裝置，與標準輸出串流 out 不同的是，err 會立即顯示指定的（錯誤）訊息給使用者知道，即使您指定期程將結果重新導向至檔案，err 輸出串流的訊息也不會被重新導向，而仍會顯示在指定的顯示裝置上，下面這個例子給您一個簡單的測試方式：

範例 3.8 ErrDemo.java

```

public class ErrDemo {
    public static void main(String[] args) {
        System.out.println("使用out輸出訊息");
        System.err.println("使用err輸出訊息");
    }
}

```

在編譯程式之後，請如下執行程式，您會發現輸出結果如下：

```
java ErrDemo > ErrDemoResult.txt
使用err輸出訊息
```

開啟 ErrDemoResult.txt 之後，您會發現當中只有 "使用out輸出訊息" 的文字，而"使用 err 輸出訊息"的文字並沒有被導向至檔案中，而是直接顯示在文字模式中。

要重新導向輸出至指定的目的檔案是用 '>'，也可以使用 '>>'，後者除了重導標準輸出之外，還有附加（Append）的功能，也就是會把輸出的內容附加到目的檔案的後頭，如果目的檔案本來不存在就會先建立一個再進行輸出。

3.2.4 輸出格式控制

標準輸出通常是使用文字模式作為輸出，這邊我介紹幾個輸出格式控制技巧，在文字模式顯示時可以協助控制輸出的格式，首先介紹格式控制字元，先使用表格列出一些常用的控制字元：

表 3.1 常用格式控制字元

控制字元	作用
\\	反斜線
'	單引號'
"	雙引號"
\uxxxx	以 16 進位數指定 Unicode 字元輸出
\xxx	以 8 進位數指定 Unicode 字元輸出
\b	倒退一個字元
\f	換頁
\n	換行
\r	游標移至行首
\t	跳格(一個Tab鍵)

範例 3.9 告訴您如何指定 Unicode 字元編碼來輸出 "Hello" 這段文字。

範例 3.9 OutputUnicode.java

```
public class OutputUnicode {
    public static void main(String[] args) {
        System.out.println("\u0048\u0065\u006C\u006C\u006F");
    }
}
```

在輸出數值時，預設都會以十進位的方式來顯示數值，範例 3.10 告訴您如何使用 java.lang.Integer 所提供的各種 toXXX() 方法來顯示不同進位制之數值。

範例 3.10 NumberDemo.java

```
public class NumberDemo {
    public static void main(String[] args) {
        // 十進位 19 轉成二進位 10011
        System.out.println(Integer.toBinaryString(19));

        // 十進位 19 轉成十六進位 13
    }
}
```

```

        System.out.println(Integer.toHexString(19));
        // 十進位 19 轉成八進位 23
        System.out.println(Integer.toOctalString(19));
    }
}

```

若是使用 J2SE 5.0 或更高的版本，您可以使用 `System.out.printf()` 作簡單的輸出格式設定，例如：

範例 3.11 TigerNumberDemo.java

```

public class TigerNumberDemo {
    public static void main(String[] args) {
        // 輸出 19 的十進位表示
        System.out.printf("%d%n", 19);

        // 輸出 19 的八進位表示
        System.out.printf("%o%n", 19);

        // 輸出 19 的十六進位表示
        System.out.printf("%x%n", 19);
    }
}

```

'%d' 表示將指定的數值以十進位表示，'%o' 是八進位表示，而 '%x' 是十六進位表示，'%n' 是指輸出平台特定的換行字元，如果是在 Windows 下實際上會置換為 '\r\n'，如果是 Linux 下則會置換為 '\n'。

表3.2 簡單列出了一些常用的轉換符號。

表 3.2 常用格式轉換字元

格式字元	作用
%%	在字串中顯示 %
%d	以 10 進位整數方式輸出，提供的數必須是 Byte、Short、Integer、Long 或 BigInteger
%f	將浮點數以 10 進位方式輸出，提供的數必須是 Float、Double 或 BigDecimal
%e, %E	將浮點數以 10 進位方式輸出，並使用科學記號，提供的數必須是 Float、Double 或 BigDecimal
%a, %A	使用科學記號輸出浮點數，以 16 進位輸出整數部份，以 10 進位輸出指數部份，提供的數必須是 Float、Double、BigDecimal
%o	以 8 進位整數方式輸出，提供的數必須是 Byte、Short、Integer、Long 或 BigInteger
%x, %X	將浮點數以 16 進位方式輸出，提供的數必須是 Byte、Short、Integer、Long 或 BigInteger
%s, %S	將字串格式化輸出
%c, %C	以字元方式輸出，提供的數必須是 Byte、Short、Character 或 Integer
%b, %B	將 "true" 或 "false" 輸出（或 "TRUE"、"FALSE"，使用 %B）。另外，非 null 值輸出是 "true"，null 值輸出是 "false"
%t, %T	輸出日期/時間的前置，詳請看線上 API 文件

您可以在輸出浮點數時指定精度，例如以下這行的執行結果會輸出 " example:19.23" :

```
System.out.printf("example:%.2f%n", 19.234);
```

您也可以指定輸出時，至少要預留的字元寬度，例如：

```
System.out.printf("example:%6.2f%n", 19.234);
```

由於預留了 6 個字元寬度，不足的部份要由空白字元補上，所以執行結果會輸出如下（19.23 只佔五個字元，所以補上一個空白在前端）：

```
example: 19.23
```

以上只是簡短的列出一些常用的輸出轉換符號，事實上，這些功能都是由 [java.util.Formatter](#) 所提供的，如果您需要更多關於輸出格式的控制，您可以看看線上 API 文件以查詢相關設定。

良葛格的話匣子 由於書的篇幅有限，我只在必要的時候列出一些常用的 API 文件表格說明，如果需要更多的說明，請直接查詢我所提供的文件索引，學會查詢線上文件也是學好 Java 的必備功夫喔！

3.3 資料、運算

電腦的原文是「Computer」，電腦的基本作用其實就是運算，要運算就要給它資料，要告訴電腦您給了它哪些資料？是整數還是小數？是文字還是字元？這就是程式語言術語中所謂的「指定變數與資料型態」！在給定了資料之後，接著您就可以叫電腦開始進行各種算術、邏輯、比較、遞增、遞減等等的運算。

3.3.1 資料型態

程式在執行的過程中，需要運算許多的資訊，也需要儲存許多的資訊，這些資訊可能是由使用者輸入、從檔案中取得，甚至是由網路上得到，在程式運行的過程中，這些資訊透過「變數」（Variable）加以儲存，以便程式隨時取用。

一個變數代表一個記憶體空間，資料就是儲存在這個空間中，使用變數名稱來取得資料相信會比使用記憶體位置來得方便；然而由於資料在儲存時所需要的容量各不相同，不同的資料必須要配給不同大小的記憶體空間來儲存，在 Java 中對不同的資料區分有幾種不同的「資料型態」（Data type）。

在 Java 中基本的資料型態（Primitive type）主要區分為「整數」、「位元」、「浮點數」、「字元」與「布林數」，而這幾種還可以細分，如下所示：

- 整數

只儲存整數數值，可細分為短整數（short）（佔 2 個位元組）、整數（int）（佔 4 個位元組）與長整數（long）（佔 8 個位元組），長整數所佔的記憶體比整數來得多，可表示的數值範圍也就較大，同樣的，整數（int）可表示的整數數值範圍也比短整數來得大。

- 位元

Java 提供有位元（byte）資料型態，可專門儲存位元資料，例如影像的位元資料，一個位元資料型態佔一個位元組，而必要的話，byte 資料型態也可以用於儲存整數數值。

- 浮點數

主要用來儲存小數數值，也可以用來儲存範圍更大的整數，可分為浮點數（float）（佔 4 個位元組）與倍精度浮點數（double）（佔 8 個位元組），倍精度浮點數所使用的記憶體空間比浮點數來得多，可表示的數值範圍與精確度也比較大。

- 字元

用來儲存字元，Java 的字元採 Unicode 編碼，其中前 128 個字元編碼與 ASCII 編碼相容；每個字元資料型態佔兩個位元組，可儲存的字元範圍由 '\u0000' 到 '\uFFFF'，由於 Java 的字元採用 Unicode 編碼，一個中文字與一個英文字母在 Java 中同樣都是用一個字元來表示。

- 布林數

佔記憶體 2 個位元組，可儲存 true 與 false 兩個數值，分別表示邏輯的「真」與「假」。因為每種資料型態所佔有的記憶體大小不同，因而可以儲存的數值範圍也就不同，例如整數（int）的記憶體空間是 4 個位元組，所以它可以儲存的整數範圍為 -2147483648 至 2147483647，如果儲存值超出這個範圍的話稱之為「溢值」（Overflow），會造成程式不可預期的結果，您可以使用範例 3.12 來獲得數值的儲存範圍。

範例 3.12 DataRange.java

```
public class DataRange {
    public static void main(String[] args) {
        System.out.printf("short \t數值範圍 : %d ~ %d\n",
                          Short.MAX_VALUE, Short.MIN_VALUE);
```

```

        System.out.printf("int \t數值範圍 : %d ~ %d\n",
                           Integer.MAX_VALUE, Integer.MIN_VALUE);
        System.out.printf("long \t數值範圍 : %d ~ %d\n",
                           Long.MAX_VALUE, Long.MIN_VALUE);
        System.out.printf("byte \t數值範圍 : %d ~ %d\n",
                           Byte.MAX_VALUE, Byte.MIN_VALUE);
        System.out.printf("float \t數值範圍 : %e ~ %e\n",
                           Float.MAX_VALUE, Float.MIN_VALUE);
        System.out.printf("double \t數值範圍 : %e ~ %e\n",
                           Double.MAX_VALUE, Double.MIN_VALUE);
    }
}

```

其中 Byte、Integer、Long、Float、Double 都是 java.lang 套件下的類別名稱，而 MAX_VALUE 與 MIN_VALUE 則是各類別中所定義的靜態常數成員，分別表示該資料型態可儲存的數值最大與最小範圍，'%e' 表示用科學記號顯示，範例的執行結果如下所示：

```

short 數值範圍 : 32767 ~ -32768
int 數值範圍 : 2147483647 ~ -2147483648
long 數值範圍 : 9223372036854775807 ~ -9223372036854775808
byte 數值範圍 : 127 ~ -128
float 數值範圍 : 3.402823e+38 ~ 1.401298e-45
double 數值範圍 : 1.797693e+308 ~ 4.900000e-324

```

其中浮點數所取得是正數的最大與最小範圍，加上負號即為負數可表示的最大與最小範圍。

3.3.2 變數、常數

資料是儲存在記憶體中的一塊空間中，為了取得資料，您必須知道這塊記憶體空間的位置，然而若使用記憶體位址編號的話相當的不方便，所以使用一個明確的名稱，變數（Variable）是一個資料儲存空間的表示，您將資料指定給變數，就是將資料儲存至對應的記憶體空間，您呼叫變數時，就是將對應的記憶體空間的資料取出供您使用。

在 Java 中要使用變數，必須先宣告變數名稱與資料型態，例如：

```

int age;           // 宣告一個整數變數
double scope;     // 宣告一個倍精度浮點數變數

```

就如上面所舉的例子，您使用 int、float、double、char 等關鍵字（Keyword）來宣告變數名稱並指定其資料型態，變數在命名時有一些規則，它不可以使用數字作為開頭，也不可以使用一些特殊字元，像是 *^% 之類的字元，而變數名稱不可以與 Java 內定的關鍵字同名，例如 int、float、class 等。

變數的命名有幾個風格，主要以清楚易懂為主，初學者為了方便，常使用一些簡單的字母來作為變數名稱，這會造成日後程式維護的困難，命名變數時發生同名的情況也會增加。在過去曾流行過匈牙利命名法，也就是在變數名稱前加上變數的資料型態名稱縮寫，例如 intNum 用來表示這個變數是 int 整數資料型態，fitNum 表示一個 float 資料型態，然而隨著現在程式的發展規模越來越大，這種命名方式已經不被鼓勵。

現在比較鼓勵用清楚的名稱來表明變數的作用，通常會以小寫字母作為開始，並在每個單字開始時第一個字母使用大寫，例如：

```

int ageOfStudent;
int ageOfTeacher;

```

像這樣的名稱可以讓人一眼就看出這個變數的作用，在 Java 程式設計領域中是比較常看到的變數命名方式。變數名稱可以使用底線作為開始，通常使用底線作為開始的變數名稱，表示它是私用的（Private），只在程式的某個範圍使用，外界並不需要知道有這個變數的存在，通常這樣的變數名稱常用於物件導向程式設計中類別的私有成員（Private member），這樣的命名方式偶而也會看到，一個宣告的例子如下：

```
double _window_center_x;
double _window_center_y;
```

當您在Java中宣告一個變數，就會配置一塊記憶體空間給它，這塊空間中原先可能就有資料，也因此變數在宣告後的值是不可預期的，Java 對於安全性的要求極高，您不可以宣告變數後，而在未指定任何值給它之前就使用它，編譯器在編譯時會回報這個錯誤，例如若宣告變數 var 却沒有指定值給它，則會顯示以下訊息：

```
variable var might not have been initialized
```

可以的話，儘量在變數宣告後初始其值，您可以使用「指定運算子」（Assignment operator）'='來指定變數的值，例如：

```
int ageOfStudent = 0;
double scoreOfStudent = 0.0;
char levelOfStudent = 'A';
```

上面這段程式在宣告變數的時候，同時指定變數的儲存值，而您也看到如何指定字元給字元變數，字元在指定時需使用引號 ' ' 來包括；在指定浮點數時，會習慣使用小數的方式來指定，如 0.0，在Java 中寫下 0.0 這麼一個常數的話，其預設為 double 資料型態。

在宣告變數之後，就可以呼叫變數名稱來取得其所儲存的值，範例 3.13 是個簡單的示範：

範例 3.13 VariableDemo.java

```
public class VariableDemo {
    public static void main(String[] args) {
        int ageOfStudent = 5;
        double scoreOfStudent = 80.0;
        char levelOfStudent = 'B';

        System.out.println("年級\t得分\t等級");
        System.out.printf("%4d\t%4.1f\t%4c",
            ageOfStudent, scoreOfStudent, levelOfStudent);
    }
}
```

以下為執行結果：

年級	得分	等級
5	80.0	B

在 Java 中寫下一個數值，這個數就稱之為字面常數（Literal constant），它會存在記憶體的某個位置，您無法改變它的值；而在使用變數的時候，您也會使用一種叫「常數」的變數，嚴格來說它並不是常數，只不過在指定數值給這個變數之後，就不可再改變其值，有人為了區分其與常數的差別，還給了它一個奇怪的名稱：「常數變數」。

先不要管「常數變數」這個怪怪的名稱，其實它終究是個變數而已，您在宣告變數名稱的同時，加上 "final" 關鍵字來限定，只不過這個變數一旦指定了值，就不可以再改變它的值，如果程式中有其它程式碼試圖改變這個變數，編譯器會先檢查出這個錯誤，例如：

```
final int maxNum = 10;
maxNum = 20;
```

這一段程式碼中的 `maxNum` 變數您使用了 "final" 關鍵字來限定，所以它在指定為 10 之後，就不可以再指定值給它，所以第二次的值指定會被編譯器指出錯誤：

```
cannot assign a value to final variable maxNum
```

使用 "final" 來限定的變數，目的通常就是不希望其它的程式碼來變動它的值，例如用於迴圈計數次數的指定（迴圈之後就會學到），或是像圓周率 PI 常數的指定。

3.3.3 算術運算

程式的目的簡單的說就是運算，除了運算還是運算，所以加減乘除這類的操作是少不得的，在 Java 中提供運算功能的就是運算子（Operator），例如與算術相關的有加（+）、減（-）、乘（*）、除（/）這類的運算子，另外還有一個也很常用的餘除運算子（%），這類以數學運算為主的運算子稱之為「算術運算子」（Arithmetic operator）

算術運算子的使用基本上與您學過的加減乘除一樣，也是先乘除後加減，必要時加上括號表示運算的先後順序，例如這個程式碼會在文字模式下顯示 7：

```
System.out.println(1 + 2 * 3);
```

編譯器在讀取程式碼時是由左往右讀取的，而初學者往往會犯一個錯誤，例如 $(1+2+3) / 4$ ，由於在數學運算上習慣將分子寫在上面，而分母寫在下面的方式，使得初學者往往將之寫成了：

```
System.out.println(1+2+3 / 4);
```

這個程式事實上會是這樣運算的： $1+2+(3/4)$ ；為了避免這樣的錯誤，在必要的時候為運算式加上括號才是最保險的，例如：

```
System.out.println((double)(1+2+3) / 4);
```

注意在上面的程式碼中使用了 `double` 限定型態轉換，如果不加上這個限定的話，程式的輸出會是 1 而不是 1.5，這是因為在這個 Java 程式中，1、2、3、4 這四個數值都是整數，當程式運算 $(1+2+3)$ 後的結果還是整數型態，若此時除以整數 4，會自動去除小數點之後的數字再進行輸出，而您加上 `double` 限定，表示要 $(1+2+3)$ 運算後的值轉換為 `double` 資料型態，如此再除以 4，小數點之後的數字才不會被去除。

同樣的，您看看這段程式會印出什麼結果？

```
int testNumber = 10;
System.out.println(testNumber / 3);
```

答案不是 3.3333 而是 3，小數點之後的部份被自動消去了，這是因為您的 `testNumber` 是整數，而除數 3 也是整數，運算出來的程式被自動轉換為整數了，為了解決這個問題，您可以使用下面的方法：

```
int testNumber = 10;
System.out.println(testNumber / 3.0);
System.out.println((double) testNumber / 3);
```

上面這個程式片段示範了兩種解決方式：如果運算式中有一個浮點數，則程式就會先轉換使用浮點數來運算，這是第一段陳述句所使用的方式；第二個方式稱之為「限定型態轉換」，您使用 `double` 告訴程式先將 `testNumber` 的值轉換為 `double`，然後再進行除法運算，所以得到的結果會是正確的 3.3333；型態轉換的限定關鍵字就是宣告變數時所使用的 `int`、`float` 等關鍵

字。

當您將精確度小的資料型態（例如int）指定給精確度大的資料型態之變數時（例如double），這樣的指定在精確度並不會失去，所以這樣的指定是可行的，由於Java對於程式的安全性要求極高，型態轉換在某些情況一定要明確指定，就是在使用指定運算子'='時，將精確度大的值指定給精確度小的變數時，由於在精確度上會有遺失，編譯器會認定這是一個錯誤，例如：

```
int testInteger = 0;
double testDouble = 3.14;
testInteger = testDouble;
System.out.println(testInteger);
```

這段程式在編譯時會出現以下的錯誤訊息：

```
possible loss of precision
found   : double
required: int
testInteger = testDouble
^
1 error
```

如果您確定要轉換數值為較小的精度，您必須明確加上轉換的限定字，編譯器才不會回報錯誤。

```
testInteger = (int) testDouble;
```

'%'運算子是餘除運算子，它計算所得到的結果是除法後的餘數，例如(10 % 3)會得到餘數1；一個使用 '%' 的例子是數字循環，假設有一個立方體要進行360度旋轉，每次要在角度上加1，而360度後必須復歸為0，然後重新計數，這時您可以這麼撰寫：

```
count = (count + 1) % 360;
```

3.3.4 比較、條件運算

數學上有比較的運算，像是大於、等於、小於等運算，Java中也提供了這些運算子，這些運算子稱之為「比較運算子」(Comparison operator)，它們有大於(>)、不小於(>=)、小於(<)、不大於(<=)、等於(==)以及不等於(!=)。

在Java中，比較的條件成立時以true表示，比較的條件不成立時以false表示，範例3.14示範了幾個比較運算的使用。

範例 3.14 ComparisonOperator.java

```
public class ComparisonOperator {
    public static void main(String[] args) {
        System.out.println("10 > 5 結果 " + (10 > 5));
        System.out.println("10 >= 5 結果 " + (10 >= 5));
        System.out.println("10 < 5 結果 " + (10 < 5));
        System.out.println("10 <= 5 結果 " + (10 <= 5));
        System.out.println("10 == 5 結果 " + (10 == 5));
        System.out.println("10 != 5 結果 " + (10 != 5));
    }
}
```

程式的執行如下所示：

```

10 > 5 結果 true
10 >= 5 結果 true
10 < 5 結果 false
10 <= 5 結果 false
10 == 5 結果 false
10 != 5 結果 true

```

比較運算在使用時有個即使是程式設計老手也可能犯的錯誤，且不容易發現，也就是等於運算子 '=='，注意它是兩個連續的等號 '=' 所組成，而不是一個等號，一個等號是指定運算，這點必須特別注意，例如若有兩個變數x與y要比較是否相等，應該是寫成 $x == y$ ，而不是寫成 $x = y$ ，後者的作用是將y的值指定給 x，而不是比較運算 x 與 y 是否相等。

另一個使用 '==' 運算時要注意的是，對於物件來說，兩個物件參考之間使用 '==' 作比較時，是比較其名稱是否參考至同一物件，而不是比較其內容，在之後的章節介紹 Integer 等包裹（Wrapper）物件或字串時會再詳細介紹。

既然談到了條件式的問題，來介紹 Java 中的「條件運算子」（conditional operator），它的使用方式如下：

條件式 ? 成立傳回值 : 失敗傳回值

條件運算子的傳回值依條件式的結果而定，如果條件式的結果為 true，則傳回冒號 ':' 前的值，若為 false，則傳回冒號後的值，範例 3.15 可以作個簡單的示範：

範例3.15 ConditionalOperator.java

```

import java.util.Scanner;

public class ConditionalOperator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("請輸入學生分數: ");
        int scoreOfStudent = scanner.nextInt();
        System.out.println("該生是否及格? " +
            (scoreOfStudent >= 60 ? '是' : '否'));
    }
}

```

這個程式會依您所輸入的分數來判斷學生成績是否不小於 60 分，以決定其是否及格，如果是則傳回字元 '是'，否則傳回字元 '否'，執行結果如下：

```

請輸入學生分數: 88
該生是否及格? 是

```

條件運算子 (?:) 使用得當的話可以少寫幾句程式碼，例如範例 3.16 可以判斷使用者輸入是否為奇數。

範例3.16 OddDecider.java

```

import java.util.Scanner;

public class OddDecider {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("請輸入數字: ");
        int number = scanner.nextInt();
        System.out.println("是否為奇數? " +
            (number%2 != 0 ? '是' : '否'));
    }
}

```

```
}
```

當您輸入的數為奇數時，就不能被2整除，所以餘數一定不是0，在條件式中判斷為 true，因而傳回字元 '是'，若數值為偶數，則2整除，所以餘數為0，在條件式中判斷為 false，所以傳回字元 '否'，一個執行的例子如下：

```
請輸入數字：87  
是否為奇數？是
```

3.3.5 邏輯、位元運算

大於、小於的運算會了，但如果想要同時進行兩個以上的條件判斷呢？例如分數大於80「且」小於90的判斷。在邏輯上有所謂的「且」（And）、「或」（Or）與「反」（Inverse），在Java中也提供這幾個基本邏輯運算所需的「邏輯運算子」（Logical operator），分別為「且」（&&）、「或」（||）及「反相」（!）三個運算子。來看看範例3.17會輸出什麼？

範例3.17 LogicalOperator.java

```
public class LogicalOperator {  
    public static void main(String[] args) {  
        int number = 75;  
        System.out.println((number > 70 && number < 80));  
        System.out.println((number > 80 || number < 75));  
        System.out.println(!(number > 80 || number < 75));  
    }  
}
```

三段陳述句分別會輸出true、false與true三種結果，分別表示number大於70「且」小於80為真、number大於80「或」小於75為假、number大於80「或」小於75的「相反」為真。

接下來看看「位元運算子」（Bitwise operator），在數位設計上有AND、OR、NOT、XOR與補數等運算，在Java中提供這些運算的就是位元運算子，它們的對應分別是&（AND）、|（OR）、^（XOR）與~（補數）。

如果您不會基本的位元運算，可以從範例3.18中瞭解各個位元運算的結果。

範例3.18 BitwiseOperator.java

```
public class BitwiseOperator {  
    public static void main(String[] args) {  
        System.out.println("AND運算：" );  
        System.out.println("0 AND 0\t\t" + (0 & 0));  
        System.out.println("0 AND 1\t\t" + (0 & 1));  
        System.out.println("1 AND 0\t\t" + (1 & 0));  
        System.out.println("1 AND 1\t\t" + (1 & 1));  
  
        System.out.println("\nOR運算：" );  
        System.out.println("0 OR 0\t\t" + (0 | 0));  
        System.out.println("0 OR 1\t\t" + (0 | 1));  
        System.out.println("1 OR 0\t\t" + (1 | 0));  
        System.out.println("1 OR 1\t\t" + (1 | 1));  
  
        System.out.println("\nXOR運算：" );  
        System.out.println("0 XOR 0\t\t" + (0 ^ 0));  
        System.out.println("0 XOR 1\t\t" + (0 ^ 1));  
        System.out.println("1 XOR 0\t\t" + (1 ^ 0));  
        System.out.println("1 XOR 1\t\t" + (1 ^ 1));  
    }  
}
```

執行結果就是各個位元運算的結果：

```

AND運算：
0 AND 0      0
0 AND 1      0
1 AND 0      0
1 AND 1      1

OR運算：
0 OR 0      0
0 OR 1      1
1 OR 0      1
1 OR 1      1

XOR運算：
0 XOR 0      0
0 XOR 1      1
1 XOR 0      1
1 XOR 1      0

```

Java 中的位元運算是逐位元運算的，例如 10010001 與 01000001 作 AND 運算，是一個一個位元對應運算，答案就是 00000001；而補數運算是將所有的位元 0 變 1，1 變 0，例如 00000001 經補數運算就會變為 11111110，例如下面這個程式所示：

```

byte number = 0;
System.out.println((int)(~number));

```

這個程式會在主控台顯示 -1，因為 byte 佔記憶體一個位元組，它儲存的 0 在記憶體中是 00000000，經補數運算就變成 11111111，這個數在電腦中用整數表示則是 -1。

要注意的是，邏輯運算子與位元運算子也是很常被混淆的，像是'&&'與'&，'||與'|'，初學時可得多注意。

位元運算對初學者來說的確較不常用，但如果用的恰當的話，可以增進不少程式效率，例如範例 3.19 可以判斷使用者的輸入是否為奇數：

範例3.19 OddDecider2.java

```

import java.util.Scanner;

public class OddDecider2 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("請輸入數字：");
        int number = scanner.nextInt();
        System.out.println("是否為奇數？" +
            ((number&1) != 0 ? '是' : '否'));
    }
}

```

執行結果：

```

請輸入數字： 66
是否為奇數？ 否

```

範例 3.19 得以運作的原理是，奇數的數值若以二進位來表示，其最右邊的位元必為 1，而偶數最右邊的位元必為 0，所以您使用 1 來與輸入的值作 AND 運算，由於 1 除了最右邊的位元為 1 之外，其它位元都會是 0，與輸入數值 AND 運算的結果，只會留下最右邊位元為 0 或為 1 的結果，其它部份都被 0 AND 運算遮掉了，這就是所謂「位元遮罩」，例如 4 與 1 作 AND 運算的結果會是 0，所以判斷為偶數：

```
整數4 : 00000100
整數1 : 00000001
AND 運算後 : 00000000
```

而 3 與 1 作 AND 運算的結果是 1，所以判斷為奇數：

```
整數3 : 00000011
整數1 : 00000001
AND 運算後 : 00000001
```

XOR 的運算較不常見，範例 3.20 舉個簡單的 XOR 字元加密例子。

範例3.20 XorCode.java

```
public class XorCode {
    public static void main(String[] args) {
        char ch = 'A';
        System.out.println("編碼前：" + ch);

        ch = (char)(ch^7);
        System.out.println("編碼後：" + ch);

        ch = (char)(ch^7);
        System.out.println("解碼：" + ch);
    }
}
```

0x7 是 Java 中整數的 16 進位寫法，其實就是 10 進位的 7，將位元與 1 作 XOR 的作用其實就是位元反轉，0x7 的最右邊三個位元為 1，所以其實就是反轉 ch 變數的最後兩個位元，如下所示：

```
ASCII 中的 'A' 字元編碼為 65 : 01000001
整數 7 : 00000111
XOR 運算後 : 01000110
```

01000110 就是整數 70，對應 ASCII 中的字元 'F' 之編碼，所以用字元方式顯示時會顯示 'F' 字元，同樣的，這個簡單的 XOR 字元加密，要解密也只要再進行相同的位元反轉就可以了，看看範例 3.20 的執行結果：

```
編碼前 : A
編碼後 : F
解碼 : A
```

良葛格的話匣子 要注意的是，我雖然在說明時都只寫下 8 個位元的值來說明，這只是為了解說方便而已。實際的位元長度在運算時，需依資料型態所佔的記憶體長度而定，例如在使用 int 型態的 0 作運算時，要考慮的是 32 個位元長度，而不是只有 8 個位元，因為 int 佔有 4 個位元組，也就是實際上是 00000000 00000000 00000000 00000000。

在位元運算上，Java 還有左移 (<<) 與右移 (>>) 兩個運算子，左移運算子會將所有的位元往左移指定的位數，左邊被擠出去的位元會被丟棄，而右邊會補上 0；右移運算則是相反，會將所有的位元往右移指定的位數，右邊被擠出去的位元會被丟棄，至於最左邊補上原來的位元，如果左邊原來是 0 就補 0，1 就補 1。另外還有 >>> 運算子，這個運算子在右移後，一定在最左邊補上 0。

範例 3.21 使用左移運算來作簡單的2次方運算示範。

範例3.21 ShiftOperator.java

```

public class ShiftOperator {
    public static void main(String[] args) {
        int number = 1;
        System.out.println("2的0次: " + number);

        number = number << 1;
        System.out.println("2的1次: " + number);

        number = number << 1;
        System.out.println("2的2次: " + number);

        number = number << 1;
        System.out.println("2的3次: " + number);
    }
}

```

執行結果：

```

2的0次: 1
2的1次: 2
2的2次: 4
2的3次: 8

```

實際來左移看看就知為何可以如此作次方運算了：

```

00000001 -> 1
00000010 -> 2
00000100 -> 4
00001000 -> 8

```

良葛格的話匣子 位元運算對於沒有學過數位邏輯的初學者來說，會比較難一些，基本上除了像是資訊工程、電機工程相關領域的開發人員會比較常使用位元運算之外，大部份的開發人員可能不常使用位元運算，如果您的專長領域比較不需要使用位元運算，則基本上先瞭解有位元運算這個東西就可以了，不必在這個部份太過鑽研。

3.3.6 遞增、遞減運算

遞增（Increment）、遞減（Decrement）與指定（Assignment）運算子，老實說常成為初學者的一個惡夢，因為有些程式中若寫得精簡，這幾個運算子容易讓初學者搞不清楚程式的真正運算結果是什麼；事實上，使用這幾種運算子的目的除了使讓程式看來比較簡潔之外，還可以稍微增加一些程式執行的效率。在程式中對變數遞增1或遞減1是很常見的運算，例如：

```

int i = 0;
i = i + 1;
System.out.println(i);
i = i - 1;
System.out.println(i);

```

這段程式會分別顯示出 1 與 0 兩個數，您可以使用遞增、遞減運算子來撰寫程式：

範例 3.22 IncrementDecrement.java

```

public class IncrementDecrement {
    public static void main(String[] args) {
        int i = 0;
        System.out.println(++i);
        System.out.println(--i);
    }
}

```

其中寫在變數 i 之前的 ++ 與 -- 就是「遞增運算子」（Increment operator）與「遞減運算子」（Decrement operator），當它們撰寫在變數之前時，其作用就相當於將變數遞增 1 與遞減 1：

```
++i;    // 相當於 i = i + 1;
--i;    // 相當於 i = i - 1;
```

您可以將遞增或遞減運算子撰寫在變數之前或變數之後，但其實兩種寫法是有差別的，將遞增（遞減）運算子撰寫在變數前時，表示先將變數的值加（減）1，然後再傳回變數的值，將遞增（遞減）運算子撰寫在變數之後，表示先傳回變數值，然後再對變數加（減）1，例如：

範例 3.23 IncrementDecrement2.java

```
public class IncrementDecrement2 {
    public static void main(String[] args) {
        int i = 0;
        int number = 0;

        number = ++i;    // 相當於 i = i + 1; number = i;
        System.out.println(number);
        number = --i;    // 相當於 i = i - 1; number = i;
        System.out.println(number);
    }
}
```

在這段程式中，number 的值會前後分別顯示為 1 與 0，再看看範例 3.24。

範例 3.24 IncrementDecrement3.java

```
public class IncrementDecrement3 {
    public static void main(String[] args) {
        int i = 0;
        int number = 0;

        number = i++;    // 相當於 number = i; i = i + 1;
        System.out.println(number);
        number = i--;    // 相當於 number = i; i = i - 1;
        System.out.println(number);
    }
}
```

在這段程式中，number 的值會前後分別顯示為 0 與 1。

接下來看「指定運算子」（Assignment operator），到目前為止您只看過一個指定運算子，也就是 '=' 這個運算子，事實上指定運算子還有以下的幾個：

表 3.3 指定運算子

指定運算子	範例	結果
<code>+=</code>	<code>a += b</code>	<code>a = a + b</code>
<code>-=</code>	<code>a -= b</code>	<code>a = a - b</code>
<code>*=</code>	<code>a *= b</code>	<code>a = a * b</code>
<code>/=</code>	<code>a /= b</code>	<code>a = a / b</code>
<code>%=</code>	<code>a %= b</code>	<code>a = a % b</code>
<code>&=</code>	<code>a &= b</code>	<code>a = a & b</code>

\	=	a \	= b	a = a \	b
$\wedge=$	$a \wedge= b$	$a = a \wedge b$			
$<<=$	$a <<= b$	$a = a << b$			
$>>=$	$a >>= b$	$a = a >> b$			

每個指定運算子的作用如上所示，但老實說若不是常寫程式的老手，當遇到這些指定運算子時，有時可能會楞一下，因為不常用的話，這些語法並不是那麼的直覺。

使用 `++`、`--` 或指定運算子，由於程式可以直接在變數的記憶體空間中運算，而不用取出變數值、運算再將數值存回變數的記憶體空間，所以可以增加運算的效率，但以現在電腦的運算速度來看，這一點的效率可能有些微不足道，除非您這類的運算相當的頻繁，否則是看不出這點效率所帶來的改善，就現在程式撰寫的規模來看，程式的易懂易讀易維護會比效率來的重要，可以的話儘量將程式寫的詳細一些會比較好，千萬不要為了賣弄語法而濫用這些運算子。

就單一個陳述而言，使用 `++`、`--` 或指定運算子是還算可以理解，但與其它陳述結合時可就得考慮一下，例如：

```
int i = 5;
arr[--i %= 10] = 10;
```

像這樣的式子，要想知道變數 `i` 是多少，以及陣列的指定索引位置在哪可就得想一下了（有興趣算一下的話，`i` 最後會是 4，而陣列的指定索引也是 4），總之，如何使用與何時使用，自己得拿捏著點。

良葛格的話匣子 在撰寫程式時，可以在運算子的左右或是逗號 `,` 之後適當的使用一些空白，讓程式看來不那麼擁擠，不僅程式看來比較美觀，閱讀起來也比較容易，比較一下就可以體會：

```
int i=0;
int number=0;
number=i++;
number=i--;
```

下面的寫法會比較好讀一些：

```
int i = 0;
int number = 0;
number = i++;
number = i--;
```

3.4 流程控制

現實生活中待解決的事千奇百怪，在電腦發明之後，想要使用電腦解決的需求也是各式各樣：「如果」發生了...，就要...；「對於」...，就一直執行...；「如果」...，就「中斷」...。為了告訴電腦在特定條件下該執行的動作，您要會使用各種條件式來定義程式執行的流程。

3.4.1 if 條件式

為了應付「如果」發生了...，就要...的需要，Java 提供了 if 條件式，它的語法如下：

```
if(條件式)
    陳述句一;
else
    陳述句二;
```

這個語法的意思，白話來說，就是當「條件式」成立時 (true)，則執行「陳述句一」，要不然 (else) 就執行「陳述句二」；如果條件式不成立時並不想作任何事，則 else 可以省略。

在 if 後如果有兩個以上陳述句，稱之為「複合陳述句」 (Compound statement)，此時必須使用 {} 定義區塊 (Block) 將複合陳述句包括起來，例如：

```
if(條件式) {
    陳述句一;
    陳述句二;
}
else {
    陳述句三;
    陳述句四;
}
```

範例 3.25 是個簡單的程式，使用if條件式來判斷使用者的輸入是奇數還是偶數。

範例 3.25 OddDecider3.java

```
import java.util.Scanner;

public class OddDecider3 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("請輸入數字: ");
        int input = scanner.nextInt();
        int remain = input % 2; // 求除 2 的餘數

        if(remain == 1) // 如果餘數為1
            System.out.println(input + "為奇數");
        else
            System.out.println(input + "為偶數");
    }
}
```

在 if 中您也可以再設定執行的條件，例如：

```
if(條件式一) {
    陳述句一;
    if(條件式二)
        陳述句二;
```

```
    陳述句三;
}
```

這只個簡單的例子，其中陳述句二要執行，必須同時滿足「條件式一」與「條件式二」才行；再來看個例子：

```
if(條件式一) {
    陳述句一;
    // 其它陳述句
}
else
    if(條件式二)
        陳述句二;
```

如果「條件式一」不滿足，就會執行 else 中的陳述，而您在這邊進行「條件式二」的測試，如果滿足就執行「陳述句二」，由於 Java 是個自由格式語言，您可以適當的排列這個程式，這會比較好懂一些：

```
if(條件式一) {
    陳述句一;
    // 其它陳述句
}
else if(條件式二)
    陳述句二;
```

基於這個方式，您可以如下設定多個條件，且易讀易懂：

```
if(條件式一)
    陳述一;
else if(條件式二)
    陳述句二;
else if(條件式三)
    陳述句三;
else
    陳述句四;
```

「陳述句四」會在條件式一、二、三都不成立時執行；範例3.26是個簡單的例子，處理學生的成績等級問題。

範例3.26 ScoreLevel.java

```
import java.util.Scanner;

public class ScoreLevel {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("輸入分數：");
        int score = scanner.nextInt();

        if(score >= 90)
            System.out.println("得A");
        else if(score >= 80 && score < 90)
            System.out.println("得B");
        else if(score >= 70 && score < 80)
            System.out.println("得C");
        else if(score >= 60 && score < 70)
            System.out.println("得D");
        else
            System.out.println("得E(不及格)");
    }
}
```

執行結果如下：

```
輸入分數 : 88
得B
```

在這邊要注意的是 if 與 else 的配對問題，例如以下程式依縮排來看，您覺得有無問題存在？

```
if(條件式一)
    if(條件式二)
        陳述句一;
    else
        陳述句二;
```

很多人都會以為「條件式二」的 if 會與 else 配對，但事實上是「條件式一」的 if 與 else 配對，加上括號的話就不會誤會了：

```
if(條件式一) {
    if(條件式二)
        陳述句一;
    else
        陳述句二;
}
```

如果想避免這種錯誤，在程式中多使用括號是必要的，多寫一些總是比少寫一些來得保險一點。

注意到了嗎？在撰寫程式時適當的使用縮排（Indent），可以在定義程式區塊時讓區塊範圍看來容易分辨。

每個開發人員使用縮排的方式各不相同，您可以使用兩個空白或是四個空白，也可以按下Tab鍵來進行縮排。

良葛格的話匣子 我習慣的縮排方式是按四次空白鍵，不習慣按 Tab 鍵，因為各個文字編輯器或 IDE 對於 Tab 字元的顯示方式都不太一樣，有些文字編輯器或 IDE 預設使用 Tab 字元來自動縮排的話，我都還會將之改為預設四個空白字元進行縮排，因為空白字元顯示方式是一致的。

3.4.2 switch 條件式

switch 只能比較數值或字元，不過別以為這樣它就比 if 來得沒用，使用適當的話，它可比 if 判斷式來得有效率；switch 的語法架構如下：

```
switch(變數名稱或運算式) {
    case 符合數字或字元:
        陳述句一;
        break;
    case 符合數字或字元:
        陳述句二;
        break;
    default:
        陳述三;
}
```

首先看看 switch 的括號，當中置放您要取出數值的變數，取出數值之後，程式會開始與 case 中所設定的數字或字元作比對，如果符合就執行當中的陳述句，直到遇到 break 後離開 switch 區塊，如果沒有符合的數值或字元，則會執行 default 後的陳述句，default 不一定需要，如果沒有預設要處理的動作，您可以省去這個部份。

來看看範例 3.26 的成績等級比對如何使用 switch 來改寫。

範例 3.27 ScoreLevel2.java

```

import java.util.Scanner;

public class ScoreLevel2 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("請輸入分數: ");
        int score = scanner.nextInt();
        int level = (int) score/10;

        switch(level) {
            case 10:
            case 9:
                System.out.println("得A");
                break;
            case 8:
                System.out.println("得B");
                break;
            case 7:
                System.out.println("得C");
                break;
            case 6:
                System.out.println("得D");
                break;
            default:
                System.out.println("得E(不及格)");
        }
    }
}

```

在這個程式中，您使用除法並取得運算後的商數，如果大於 90 的話，除以 10 的商數一定是 9 或 10（100 分時），在 case 10 中沒有任何的陳述，也沒有使用 break，所以會繼續往下執行，直到遇到 break 離開 switch 為止，所以學生成績 100 分的話，也會顯示 A 的成績等級；如果比對的條件不在 10 到 6 這些值的話，會執行 default 下的陳述，這表示商數小於 6，所以學生的成績等級就顯示為 E 了。

注意在 case 後的等號是冒號而不是分號，這是個很常打錯的符號；如果您比對的是字元，則記得加上單引號（'」），例如：

```
case 'A':
```

這個程式與使用 if 來判斷成績等級的程式有何不同？如果純粹比對數字或字元的話，建議使用 switch，因為它只會在一開始的 switch 括號中取出變數值一次，然後將這個值與下面所設定的 case 比對，但如果使用 if 的話，每次遇到條件式時，都要取出變數值，效率的差異就在這，例如：

```

if(a == 1)
    // ...
else if(a == 2)
    // ...
else if(a == 3)
    // ...

```

這個程式片段在最差的狀況下，也就是 a = 3 時，共需三次比對，而每次比對都必須取出變數 a 的值一次，如果換成 switch 的話：

```

switch(a) {
    case 1:
        // ..
        break;
    case 2:
        // ..
        break;
    case 3:
        // ..
        break;
}

```

在這個程式片段中，您只在一開頭 switch 的括號中取出變數 a 的值，然後逐一比對下面的 case，效率的差別就在於這邊；當然並不是使用 if 就不好，遇到複合條件時，switch 就幫不上忙了，您無法在 switch 中組合複雜的條件陳述，這時就得使用 if 了，簡單的說，if 與 switch 兩者可以搭配著靈活使用。

3.4.3 for 迴圈

在 Java 中如果要進行重複性的指令執行，可以使用 for 迴圈式，它的基本語法如下：

```
for(初始式; 判斷式; 遞增式) {
    陳述句一;
    陳述句二;
}
```

如果陳述句只有一個，也就是非複合陳述句，{} 可以省略不寫；for 迴圈的「第一個初始陳述句只會執行一次」，之後每次重新進行迴圈時，都會「根據判斷式來判斷是否執行下一個迴圈」，而每次執行完迴圈之後，都會「執行遞增式」。

實際看看範例 3.28 來瞭解 for 迴圈的功能。

範例3.28 SimpleLoopFor.java

```
public class SimpleLoopFor {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++)
            System.out.print(" " + i);
    }
}
```

執行結果：

```
0 1 2 3 4 5 6 7 8 9
```

這是一個簡單的例子，但說明 for 的作用再適合不過，在 Java 中您可以直接在 for 中宣告變數與指定初始值，這個宣告的變數在 for 迴圈結束之後也會自動消失；初始變數的陳述句只被執行一次，接下來迴圈會根據 i 是否小於 10 來判斷是否執行迴圈，而每執行一次迴圈就將 i 加 1。

for 迴圈中也可以再使用 for 迴圈，初學者很喜歡用的例子就是顯示九九乘法表，這邊就用這個例子來說明：

範例3.29 NineTable.java

```
public class NineTable {
    public static void main(String[] args) {
        for(int j = 1; j < 10; j++) {
            for(int i = 2; i < 10; i++) {
                System.out.printf("%d*%d=%2d ", i, j, i * j);
            }
            System.out.println();
        }
    }
}
```

執行結果：

```

2*1= 2   3*1= 3   4*1= 4   5*1= 5   6*1= 6   7*1= 7   8*1= 8   9*1= 9
2*2= 4   3*2= 6   4*2= 8   5*2=10  6*2=12  7*2=14  8*2=16  9*2=18
2*3= 6   3*3= 9   4*3=12  5*3=15  6*3=18  7*3=21  8*3=24  9*3=27
2*4= 8   3*4=12  4*4=16  5*4=20  6*4=24  7*4=28  8*4=32  9*4=36
2*5=10  3*5=15  4*5=20  5*5=25  6*5=30  7*5=35  8*5=40  9*5=45
2*6=12  3*6=18  4*6=24  5*6=30  6*6=36  7*6=42  8*6=48  9*6=54
2*7=14  3*7=21  4*7=28  5*7=35  6*7=42  7*7=49  8*7=56  9*7=63
2*8=16  3*8=24  4*8=32  5*8=40  6*8=48  7*8=56  8*8=64  9*8=72
2*9=18  3*9=27  4*9=36  5*9=45  6*9=54  7*9=63  8*9=72  9*9=81

```

事實上，for 迴圈的語法其實只是將三個複合陳述區塊寫在一個括號中而已，所不同的是第一個陳述區塊只會執行一次，第二個陳述區塊專司判斷是否繼續下一個迴圈，而第三個陳述區塊只是一般的陳述句，而不一定只放遞增運算式。

for 括號中的每個陳述區塊是以分號 ';' 作區隔，而在一個陳述區塊中若想寫兩個以上的陳述句，則使用逗號 ',' 作區隔，有興趣的話，看看範例 3.30 中九九乘法表的寫法，它只使用了一個 for 迴圈就可以完成九九乘法表的列印，執行結果與範例 3.29 是一樣的。

範例 3.30 NineTable2.java

```

public class NineTable2 {
    public static void main(String[] args) {
        for (int i = 2, j = 1;
             j < 10;
             i = (i==9)?((++j/j)+1):(i+1)) {
            System.out.printf("%d%d=%2d%c", i, j, i * j,
                             (i==9 ? '\n' : ' '));
        }
    }
}

```

這個程式的重點在於讓您看看 for 迴圈的括號中之寫法，當然原則上並不鼓勵這麼寫，程式基本上還是以清楚易懂為原則，至於這個九九乘法表的演算其實也不難，當中的語法之前都介紹過了，您可以親自演算看看就知道怎麼回事了。

3.4.4 while 迴圈

Java 提供 while 迴圈式，它根據您所指定的條件式來判斷是否執行迴圈本體，語法如下所示：

```

while(條件式) {
    陳述句一;
    陳述句二;
}

```

如果迴圈本體只有一個陳述句，則 while 的 {} 可以省略不寫；while 像是沒有起始陳述與終止陳述的 for 迴圈，主要用於重複性的動作，而停止條件未知何時發生的情況，例如一個使用者輸入介面，使用者可能輸入 10 次，也可能輸入 20 次，這時迴圈停止的時機是未知的，您可以使用 while 迴圈來作這個事。

一個計算輸入成績平均的程式如範例 3.31 所示。

範例 3.31 ScoreAverage.java

```

import java.util.Scanner;

public class ScoreAverage {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int score = 0;
        int sum = 0;
        int count = -1;
    }
}

```

```

        while(score != -1) {
            count++;
            sum += score;
            System.out.print("輸入分數(-1結束) : ");
            score = scanner.nextInt();
        }

        System.out.println("平均：" + (double) sum/count);
    }
}

```

在這個程式中，使用者的輸入次數是未知的，所以您使用 while 迴圈來判斷使用者的輸入是否為 -1，以作為迴圈執行的條件，一個執行的例子如下：

```

輸入分數(-1結束) : 99
輸入分數(-1結束) : 88
輸入分數(-1結束) : 77
輸入分數(-1結束) : -1
平均 : 88.0

```

while 可以用作無窮迴圈，很多地方都用到的到無窮迴圈，例如遊戲設計中對使用者輸入裝置的輪詢（poll），或是動畫程式的播放都會使用到無窮迴圈，一個無窮迴圈如下所示：

```

while(true) {
    // 遷圈內容;
    ...
}

```

無窮迴圈可以由自己迴圈中的某個條件式來結束，例如下面是一個迴圈內部終止的例子：

```

while(true) {
    // 陳述句;
    if(條件式)
        break; // 跳離迴圈
    ...
}

```

當「條件式」成立時，會執行 break 離開 while 迴圈，這個 break 與 switch 中的作用是一樣的，都是要離開當時執行的程式區塊時使用。

while 迴圈有時稱之為「前測式迴圈」，因為它在迴圈執行前就會進行條件判斷，而另一個 do while 稱之「後測式迴圈」，它會先執行迴圈本體，然後再進行條件判斷，do while 的語法如下所示：

```

do {
    // 陳述句一;
    // 陳述句二;
    ...
} while(條件式);

```

注意 while 後面是以分號 ';' 作為結束，這個很常被忽略；由於 do while 會先執行迴圈，所以它通常用於進行一些初始化或介面溝通的動作，例如範例 3.32 所示。

範例 3.32 OddDecider4.java

```
import java.util.Scanner;
```

```

public class OddDecider4 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int input = 0;
        int replay = 0;

        do {
            System.out.print("輸入整數值：");
            input = scanner.nextInt();
            System.out.println("輸入數為奇數？" +
                ((input%2 == 1) ? 'Y': 'N'));
            System.out.print("繼續(1:繼續 0:結束)？");
            replay = scanner.nextInt();
        } while(replay == 1);
    }
}

```

執行結果：

```

輸入整數值：77
輸入數為奇數？Y
繼續(1:繼續 0:結束)？0

```

3.4.5 break、continue

break 可以離開目前 switch、for、while、do while 的區塊，並前進至區塊後下一個陳述句，在 switch 中主要用來中斷下一個 case 的比對，在 for、while 與 do while 中，主要用於中斷目前的迴圈執行，break 的例子您之前已經看過不少，這邊不再舉例。

continue 的作用與 break 類似，主要使用於迴圈，所不同的是 break 會結束區塊的執行，而 continue 只會結束其之後區塊的陳述句，並跳回迴圈區塊的開頭繼續下一個迴圈，而不是離開迴圈，例如：

```

for(int i = 1; i < 10; i++) {
    if(i == 5)
        break;

    System.out.println("i = " + i);
}

```

這段程式會顯示 $i = 1$ 到 $i = 4$ ，因為當 i 等於 5 時就會執行 break 而離開迴圈，再看下面這個程式：

```

for(int i = 1; i < 10; i++) {
    if(i == 5)
        continue;

    System.out.println("i = " + i);
}

```

這段程式會顯示 1 到 4，與 6 到 9，當 i 等於 5 時，會執行 continue 直接結束此次迴圈，這次迴圈中 System.out.println() 該行並沒有被執行，然後從區塊開頭執行下一次迴圈，所以 5 並沒有被顯示。

break 與 continue 還可以配合標籤使用，例如本來 break 只會離開 for 迴圈，設定標籤與區塊，則可以離開整個區塊，範例 3.33 舉個簡單的示範。

範例 3.33 BreakTest.java

```

public class BreakTest {
    public static void main(String[] args) {

```

```

back : {
    for(int i = 0; i < 10; i++) {
        if(i == 9) {
            System.out.println("break");
            break back;
        }
    }
    System.out.println("test");
}
}
}

```

程式的執行結果會顯示 break ; back 是個標籤，當 break back; 時，返回至 back 標籤處，之後整個 back 區塊不執行而跳過，所以這個程式 System.out.println("test"); 不會被執行。

事實上 continue 也有類似的用法，只不過標籤只能設定在 for 之前，範例 3.34 舉個簡單的示範。

範例3.34 ContinueTest.java

```

public class ContinueTest {
    public static void main(String[] args) {
        back1:
        for(int i = 0; i < 10; i++){
            back2:
            for(int j = 0; j < 10; j++) {
                if(j == 9) {
                    continue back1;
                }
            }
            System.out.println("test");
        }
    }
}

```

continue 配合標籤，可以自由的跳至任何一層 for 迴圈，您可以試試 continue back1 與 continue back2 的不同，設定 back1 時，System.out.println("test"); 不會被執行。

3.5 接下來的主題

每一個章節的內容由淺至深，初學者該掌握的深度要到哪呢？在這個章節中，對於初學者我建議至少掌握以下幾點內容：

- 能輸出訊息至文字模式
- 能從文字模式下取得使用者的輸入
- 懂得使用表 3.1、表 3.2 的格式字元
- 得 Java 中的基本資料型態與變數宣告
- 會使用各種運算子，位元運算的話視您的領域而定，如果不常用的話可以暫時忽略
- 多使用 if、for、switch、while 作程式練習

程式語言本質上就是語言，就像中文、英文一樣，要學習語言必先學習文法，這個章節所介紹的就是 Java 語言的基本文法（語法），接下來就是多使用這種語言以求熟悉這個語言，多找些題目來練習是必要的，在練習的同時培養使用 Java 語言的邏輯來思考問題，您可以在我的網站上找到一些常見的程式題目來作練習：

- <http://openhome.cc/Gossip/AlgorithmGossip/>

接下來我會介紹 Java SE 下提供的一些標準物件如何使用，在您正式瞭解 Java 的物件導向程式設計之前，您可以先將物件理解為一個現成的工具，Java SE 下提供了許多標準工具可以使用，熟悉這些工具的使用也是學習 Java 的必備功課。

第 4 章 從 **autoboxing、unboxing** 認識物件

在使用 Java 語言撰寫程式時，幾乎都是在處理「物件」（Object），您可以將物件當作一個具體的「工具」，在真正開始學習 Java 的物件導向設計之前，可以先從學習如何使用 Java SE 提供的種種工具開始，然而在第 3 章中學習到基本型態（Primitive type），這些在 J2SE 5.0 前預設並不是以物件的形式存在，您必須親自將之包裹為物件，然後才能像物件一樣的操作它。

在 J2SE 5.0 開始為基本型態提供了自動裝箱（autoboxing）、拆箱（unboxing）的功能，讓您在將基本型態轉換為物件時更為方便，而從中您也可以體會到基本型態與物件的差別。

4.1 關於物件

基本型態 long、int、double、float、boolean 等，在 J2SE 5.0 之前必須親自使用 Long、Integer、Double、Float、Boolean 等類別將之包裹為物件，如此才能將之當作物件來操作，即使 J2SE 5.0 開始支援了自動裝箱（autoboxing）、拆箱（unboxing），您仍然有必要瞭解如何親自包裹基本型態，這有助於您瞭解物件與基本型態的差別。

4.1.1 使用物件

在 Java 中會經常談到「類別」（Class）與「物件」（Object）這兩個名詞，要詳細談這兩個名詞的差別，就要詳細討論物件導向程式設計的內容，這邊先給您簡單的解釋：「類別就像是物件的設計圖，它定義了物件可操作的功能。」

不以 Java 的術語而以白話來說明的話，物件就像是一件具體的工具，而類別定義了這個工具如何產生。例如您有一張剪刀的設計圖，並依這張設計圖製作了一隻剪刀，則設計圖就類似於我所說的類別，而製作出來的剪刀就類似於我說的工具。

在正式開始學會 Java 物件導向程式設計之前，您要先學會如何使用 J2SE 所提供的一系列標準工具（也就是標準物件），您指明所要使用的設計圖（也就是類別）來產生具體的工具（也就是物件），工具的設計圖內容實際為何您不用擔心，您所要作的就是操作這些工具並配合第 3 章中介紹的流程控制語法，以完成您所需的程式。

舉個簡單的例子，如果您想寫一個程式取得現在的系統時間，您只要產生一個 java.util.Date 工具就可以了，至於 Date 實際上如何向系統取得時間，則無需您來操心，範例 4.1 示範如何取得系統時間。

範例 4.1 NowTime.java

```
import java.util.Date;

public class NowTime {
    public static void main(String[] args) {
        Date date = new Date();
        System.out.println(date.toString());
    }
}
```

Date date 表示您指定的工具為 Date 類型，而參考名稱（reference name）為 "date"；new Date() 表示您要程式產生這麼一個 Date 工具。您可以將參考名稱想像成一個名牌，它現在綁在一個 Date 工具上。在取得 Date 的實例（Instance）之後，您就可以透過參考名稱來操作它。這就好比公司員工身上會有個"跑腿"名牌，名牌掛在哪個員工身上，透過"跑腿"名牌呼叫時，有名牌的那個員工就要跑腿。

範例 4.1 中操作 `toString()` 方法，要求 Date 的實例產生目前的系統時間，然後您將之輸出在文字模式下（有關 Date 操作的詳細說明，請見 18 章），執行結果如下：

```
Tue May 03 16:06:46 GMT+08:00 2005
```

另一個操作物件最顯而易見的就是字串的操作，在 Java 中字串就是物件，是 `java.lang.String` 類別的一個實例，我會在第 6 章詳細介紹字串的特性，在這邊我先以範例 4.2 展現幾個簡單的字串操作。

範例 4.2 StringDemo.java

```
public class StringDemo {
    public static void main(String[] args) {
        String text = "Have a nice day!! :)";
        System.out.println("原文：" + text);
    }
}
```

```
// 傳回全為大寫的字串內容
System.out.println("大寫：" + text.toUpperCase());

// 轉回全為小寫的字串內容
System.out.println("小寫：" + text.toLowerCase());

// 計算字串長度
System.out.println("長度：" + text.length());

// 傳回取代文字後的字串
System.out.println("取代：" + text.replaceAll("nice", "good"));

// 傳回指定位置後的子字串
System.out.println("子字串：" + text.substring(5));
}

}
```

在程式中的每一個陳述都已經說明了每個方法的作用，直接來看執行的結果：

```
原文：Have a nice day!! :)
大寫：HAVE A NICE DAY!! :)
小寫：have a nice day!! :)
長度：20
取代：Have a good day!! :)
子字串：a nice day!! :)
```

在 Java 中直接使用 "" 包括的字串就是一個物件，範例 4.3 展現了一個簡單的使用者登入程式，運用了 String 物件的 equals() 方法。

範例 4.3 UserLogin.java

```
import java.util.Scanner;

public class UserLogin {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("使用者名稱：");
        String username = scanner.next();

        System.out.print("使用者密碼：");
        String password = scanner.next();

        if("caterpillar".equals(username)
            && "1975".equals(password)) {

            System.out.println("秘密資訊在此！");
        }
        else {
            System.out.println(username +
                " 您好，輸入的登入資料有誤，請重新輸入！");
        }
    }
}
```

String 物件的 equals() 方法可以用來比對指定的字串是否有相同的字元內容，如果相同就傳回 true，不然就傳回 false，以下是執行的結果：

```
使用者名稱：caterpillar
使用者密碼：1975
秘密資訊在此！
```

在範例 4.3 中，您還使用了 Scanner 物件來協助您取得使用者字串的輸入，雖然您並不知道 Scanner 物件實際上如何取得輸

入，物件本身包括了如何取得資訊方式，但對您隱藏這些資訊，您只要透過它所提供的方法，就可以完成相對應的操作。

4.1.2 包裹 (Wrap) 基本型態

回過頭來看看基本型態：long、int、double、float、boolean、byte 等。在 Java 中這些並不是物件，它只是純粹的資料，除了數值本身的資訊之外，基本型態不帶有其它的資訊或可操作的方法。

您已經看過幾個操作物件的範例，也知道使用物件的好處，物件本身可以攜帶更多的資訊，所以如果基本型態可以物件的形式存在，它可以帶有更多的訊息並提供相對應的操作方法，在 J2SE 5.0 之前，如果您想要讓基本型態能像物件一樣操作，那麼您要使用 Long、Integer、Double、Float、Boolean、Byte 等類別來包裹 (Wrap) 基本型態。Long、Integer、Double、Float、Boolean 等類別是所謂的 Wrapper 類別，正如其名稱所表示的，這些類別的主要目的，就是讓您提供一個物件實例作為「殼」，將基本型態包到這個物件之中，如此您就可以操作這個物件，就好像您將基本型態當作物件一樣操作，您可以直接看看範例 4.4 來瞭解體會。

範例 4.4 WrapperDemo.java

```
public class WrapperDemo {
    public static void main(String[] args) {
        int data1 = 10;
        int data2 = 20;

        // 使用Integer來包裹int資料
        Integer data1Wrapper = new Integer(data1);
        Integer data2Wrapper = new Integer(data2);

        // 直接除以3
        System.out.println(data1 / 3);

        // 轉為double值再除以3
        System.out.println(data1Wrapper.doubleValue() / 3);

        // 進行兩個值的比較
        System.out.println(data1Wrapper.compareTo(data2Wrapper));
    }
}
```

在第 2 章中有提到過，如果您將兩個整數進行相除，預設上會將小數點後的資料去除，而在範例 4.4 中，您將整數使用 Integer 包裹，並使用它的 doubleValue() 傳回 double 值，如此再進行相除時小數點後就不會被去除，而 Integer 上也提供 compareTo() 可以直接比較與另一個 Integer 物件是否相等，如果是就傳回 0，比指定值小的話傳回 -1，比指定值大的話傳回 1，與使用 '==' 只能比較是否相同，compareTo() 方法傳回更多的資訊，執行的結果如下：

```
3
3.3333333333333335
-1
```

依照同樣的方式，您也可以將 long、double、float、boolean、byte 等，使用對應的 Long、Double、Float、Boolean、Byte 等類別進行包裹，之後就可以進行物件操作。

良葛格的話匣子 在 IDE 上撰寫程式的話，資料以物件方式存在也可以提高寫程式的效率，因為大部份的 IDE 對物件都會提示可使用的方法，如此在選取物件方法時會很有幫助，例如在 Eclipse 上的一個示範畫面如下：

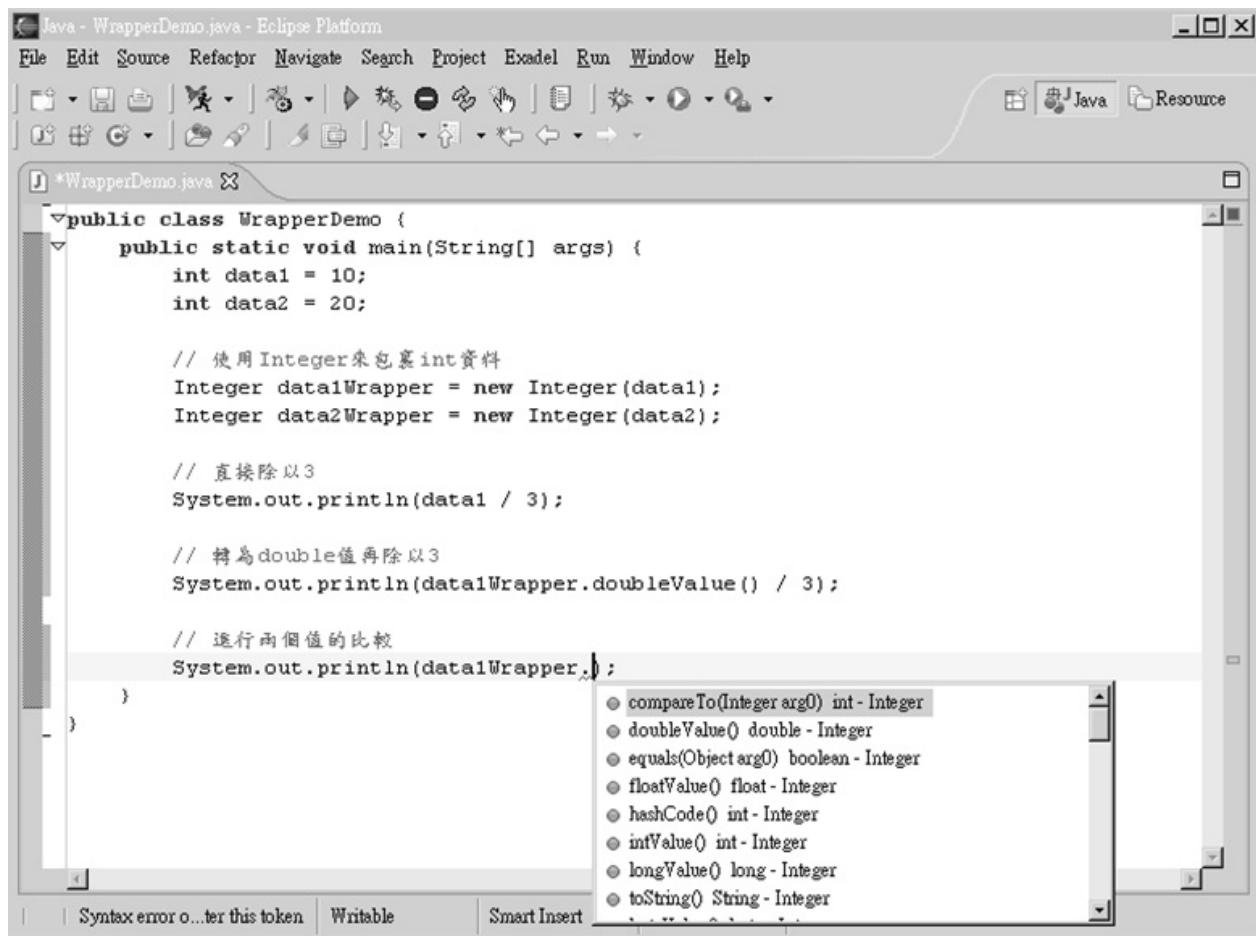


圖 4.1 Eclipse 上的物件方法提示

4.2 自動裝箱、拆箱

基本（Primitive）型態的自動裝箱（autoboxing）、拆箱（unboxing）是 J2SE 5.0 提供的新功能，雖然提供了您不用自行包裹基本型態的方便性，但提供方便的同時表示隱藏了細節，建議您在能夠區分基本型態與物件的差別時再來使用。

4.2.1 autoboxing、unboxing

在 Java 中，所有您要處理的東西「幾乎」都是物件（Object），例如您之前所使用的 Scanner 是物件，字串（String）也是物件，您之後還會看到更多的物件，然而基本（Primitive）資料型態不是物件，也就是您使用 int、double、boolean 等宣告的變數，以及您在程式中直接寫下的字面常量。

在前一個小節中您已經大致看過操作物件的方便性，而使用 Java 有一段時間的人都知道，有些時候您需要將基本型態轉換為物件，例如使用 Map 物件要操作 put() 方法時，需要傳入的引數是物件而不是基本型態。

您要使用包裹型態（Wrapper Types）才能將基本資料型態包裝為物件，前一個小節中您已經知道在 J2SE 5.0 之前，要如下才能將 int 包裝為一個 Integer 物件：

```
Integer integer = new Integer(10);
```

在 J2SE 5.0 之後提供了自動裝箱的功能，您可以直接如下撰寫來包裹基本型態：

```
Integer integer = 10;
```

在進行編譯時，編譯器在自動根據您寫下的陳述，判斷是否為您進行自動裝箱動作，在上例中您的 integer 參考的會是 Integer 類別的實例；同樣的動作可以適用於 boolean、byte、short、char、long、float、double 等基本型態，分別會使用對應的包裹型態（Wrapper Types）Boolean、Byte、Short、Character、Integer、Long、Float 或 Double；直接使用自動裝箱功能來改寫一下範例 4.4 作為練習。

範例 4.5 AutoBoxDemo.java

```
public class AutoBoxDemo {
    public static void main(String[] args) {
        Integer data1 = 10;
        Integer data2 = 20;

        // 轉為double值再除以3
        System.out.println(data1.doubleValue() / 3);

        // 進行兩個值的比較
        System.out.println(data1.compareTo(data2));
    }
}
```

程式看來簡潔了許多，data1 與 data2 在運行時就是 Integer 的實例，可以直接進行物件操作，執行的結果如下：

```
3.3333333333333335
-1
```

自動裝箱運用的方法還可以如下：

```
int i = 10;
```

```
Integer integer = i;
```

您也可以使用更一般化的 `java.lang.Number` 類別來自動裝箱，例如：

```
Number number = 3.14f;
```

`3.14f` 會先被自動裝箱為 `Float`，然後指定給 `number`。

J2SE 5.0 中可以自動裝箱，也可以自動拆箱（unboxing），也就是將物件中的基本形態資訊從物件中自動取出，例如下面這樣寫是可以的：

```
Integer fooInteger = 10;
int fooPrimitive = fooInteger;
```

`fooInteger` 參考至自動裝箱為 `Integer` 的實例後，如果被指定給一個 `int` 型態的變數 `fooPrimitive`，則會自動變為 `int` 型態再指定給 `fooPrimitive`；在運算時，也可以進行自動裝箱與拆箱，例如：

```
Integer i = 10;
System.out.println(i + 10);
System.out.println(i++);
```

上例中會顯示 20 與 10，編譯器會自動幫您進行自動裝箱與拆箱，也就是 10 會先被裝箱，然後在 `i + 10` 時會先拆箱，進行加法運算；`i++` 該行也是先拆箱再進行遞增運算。再來看一個例子：

```
Boolean boo = true;
System.out.println(boo && false);
```

同樣的 `boo` 原來是 `Boolean` 的實例，在進行 AND 運算時，會先將 `boo` 拆箱，再與 `false` 進行 AND 運算，結果會顯示 `false`。

4.2.2 小心使用 boxing

自動裝箱與拆箱的功能事實上是編譯器來幫您的忙，編譯器在編譯時期依您所撰寫的語法，決定是否進行裝箱或拆箱動作，例如：

```
Integer i = 100;
```

相當於編譯器自動為您作以下的語法編譯：

```
Integer i = new Integer(100);
```

所以自動裝箱與拆箱的功能是所謂的「編譯器蜜糖」（Compiler sugar），雖然使用這個功能很方便，但在程式運行階段您還是瞭解 Java 的語義，例如下面的程式是可以通過編譯的：

```
Integer i = null;
int j = i;
```

這樣的語法在編譯時期是合法的，但是在運行時期會有錯誤，因為這種寫法相當於：

```
Integer i = null;
int j = i.intValue();
```

null 表示 i 沒有參考至任何的物件實體，它可以合法的指定給物件參考名稱，由於實際上 i 並沒有參考至任何的物件，所以也就不可能操作 intValue() 方法，所以上面的寫法在運行時會出現 NullPointerException 的錯誤。

自動裝箱、拆箱的功能提供了方便性，但隱藏了一些細節，所以必須小心，再來看範例 4.6，您以為結果是如何？

範例 4.6 AutoBoxDemo2.java

```
public class AutoBoxDemo2 {
    public static void main(String[] args) {
        Integer i1 = 100;
        Integer i2 = 100;

        if (i1 == i2)
            System.out.println("i1 == i2");
        else
            System.out.println("i1 != i2");
    }
}
```

以自動裝箱與拆箱的機制來看，您可能會覺得結果是顯示 "i1 == i2"，您是對的！那麼範例 4.7 的這個程式，您覺得結果是什麼？

範例 4.7 AutoBoxDemo3.java

```
public class AutoBoxDemo3 {
    public static void main(String[] args) {
        Integer i1 = 200;
        Integer i2 = 200;

        if (i1 == i2)
            System.out.println("i1 == i2");
        else
            System.out.println("i1 != i2");
    }
}
```

結果是顯示 "i1 != i2"，這有些令人訝異，兩個範例語法完全一樣，只不過改個數值而已，結果卻相反。

其實這與 '==' 運算子的比較有關，在第 3 章中介紹過 '==' 是用來比較兩個基本型態的變數值是否相等，事實上 '==' 也用於判斷兩個物件參考名稱是否參考至同一個物件。

在自動裝箱時對於值從 -128 到 127 之間的值，它們被裝箱為 Integer 物件後，會存在記憶體之中被重用，所以範例 4.6 中使用 '==' 進行比較時，i1 與 i2 實際上參考至同一個物件，如果超過了從 -128 到 127 之間的值，被裝箱後的 Integer 物件並不會被重用，即相當於每次裝箱時都新建一個 Integer 物件，所以範例 4.7 使用 '==' 進行比較時，i1 與 i2 參考的是不同的物件。

所以不要過份依賴自動裝箱與拆箱，您還是必須知道基本型態與物件的差異，範例 4.7 最好還是依正規的方式來寫，而不是依賴編譯器蜜糖（Compiler sugar），例如範例 4.7 必須改寫為範例 4.8 才是正確的。

範例 4.8 AutoBoxDemo4.java

```
public class AutoBoxDemo4 {  
    public static void main(String[] args) {  
        Integer i1 = 200;  
        Integer i2 = 200;  
  
        if (i1.equals(i2))  
            System.out.println("i1 == i2");  
        else  
            System.out.println("i1 != i2");  
    }  
}
```

結果這次是顯示 "i1 == i2" 了，使用這樣的寫法，相信您也會比較放心一些，對於這些方便但隱藏細節的功能到底要不要用呢？基本上只有一個原則：如果您不確定就不要用。

良葛格的話匣子 基本上我是建議新手不要使用自動裝箱、拆箱的語法，在這邊說明這個功能是為了要完整性介紹 J2SE 5.0 的特性，新手入門的話，最好在對物件有較深入瞭解之後，再來使用這個功能。

4.3 接下來的主題

每一個章節的內容由淺至深，初學者該掌握的深度要到哪呢？在這個章節中，對於初學者我建議至少掌握以下幾點內容：

- 瞭解基本型態與物件型態的差異 接下來的章節要介紹的大部份就都要接觸物件了，像是陣列、字串等，在Java中都是以物件的方式存在，您可以從中瞭解直接操作物件的好處，並在實際操作物件的過程中逐步認識物件。

第 5 章 陣列

陣列（Array）本質上是一組資料的群組，每筆資料會有一個索引值（Index），您只要指定索引值就可以取出對應的資料，在程式中經常會使用陣列進行資料的整理與存取；在 Java 中，陣列不僅僅是一組資料群組，當您宣告一個陣列時，也就是在生成一個陣列物件，將陣列當作物件來操作，比傳統上的一些程式語言只將陣列當作資料群組多了不少好處。

這個章節會對陣列與物件之間的關係作深入的討論，如果您之前對陣列的認知只是一組資料的話，請在這一個章節中改變這個觀念；另外我也將介紹 J2SE 5.0 中對 Arrays 類別的功能加強，以及 J2SE 5.0 新增的 foreach 語法（Enhanced for loop），看看如何使用 foreach 語法來簡化陣列循序存取時的撰寫方式。

5.1 一維陣列、二維陣列

不管在其它語言中是如何，陣列在 Java 中可得看作一個物件，它有一些值得探討的特性，這個小節會先介紹最常使用的一維陣列與二維陣列。

5.1.1 一維陣列物件

您現在要整理全班的 Java 小考成績，您希望寫個小程序，全班共有 40 名學生，所以您必須有 40 個變數來儲存學生的成績，現在問題來了，根據第 3 章學過的變數宣告方式，難道您要宣告 40 個名稱不同的變數來儲存學生的成績資料嗎？

當然不必這麼麻煩，Java 提供「陣列」（Array）讓您可以宣告一個以「索引」（Index）作為識別的資料結構，在 Java 中，您可以這麼宣告一個陣列並初始陣列內容：

```
int[] score = {90, 85, 55, 94, 77};
```

這個程式片段宣告了一個 score 陣列，它的內容包括 90、85、55、94 與 77 這五個元素，您要存取陣列時，必須使用索引值來指定存取陣列中的哪個元素，在 Java 中陣列的索引是由 0 開始，也就是說索引 0 的位置儲存 90、索引 1 的位置儲存 85、索引 2 的位置儲存 55，依此類推，如果您要循序的取出陣列中的每個值並顯示出來，您可以使用 for 迴圈，如範例 5.1 所示。

範例 5.1 SimpleArray.java

```
public class SimpleArray {
    public static void main(String[] args) {
        int[] score = {90, 85, 55, 94, 77};

        for(int i = 0; i < score.length; i++)
            System.out.printf("score[%d] = %d\n", i, score[i]);
    }
}
```

在範例 5.1 中，在每次的 i 遞增後，都會作為陣列的索引指定以取出對應的陣列值，執行的結果如下：

```
score[0] = 90
score[1] = 85
score[2] = 55
score[3] = 94
score[4] = 77
```

在存取陣列元素時，必須注意到您指定的索引值不可超出陣列範圍，例如在範例 5.1 中，陣列最多可以索引到 4，所以您不可以存取超過 4 的索引值，否則會發生 `ArrayIndexOutOfBoundsException` 例外，如果您不處理這個例外，程式將會終止，第 10 章會詳細說明例外處理的方法。

範例 5.1 中使用了 `length` 這個陣列物件的屬性成員，在 Java 中陣列是一個物件，而不是單純的資料集合，陣列物件的 `length` 屬性成員可以收回陣列的長度，也就是陣列中的元素個數。

當您宣告一個陣列時，其實就是在配置一個陣列物件，實際上範例 5.1 中示範的只是陣列宣告與初始化成員的一個簡易宣告方式，在 Java 中物件都是以 `new` 來配置記憶體空間，陣列的使用也不例外，事實上一個完整的陣列宣告方式如下所示：

```
int[] arr = new int[10];
```

在上面的宣告中，arr 是個 int[] 型態的參考名稱，程式會為 arr 配置可以儲存 10 個 int 整數的一維陣列物件，索引為 0 到 9，初始值預設為 0，在 Java 中配置陣列之後，若還沒有指定初值，則依資料型態的不同，會預設有不同的初值，如表 5.1 所示。

表 5.1 陣列元素初始值

資料型態	初始值
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	\u0000
boolean	false

範例 5.2 使用配置的語法來宣告陣列，並使用 for 迴圈來設定每個元素的值然後顯示出來。

範例 5.2 ArrayDemo.java

```
public class ArrayDemo {
    public static void main(String[] args) {
        int[] arr = new int[10];

        System.out.print("arr 初始值: ");
        for(int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
            arr[i] = i;
        }

        System.out.print("\narr 設定值: ");
        for(int i = 0; i < arr.length; i++)
            System.out.print(arr[i] + " ");
        System.out.println();
    }
}
```

執行結果：

```
arr 初始值: 0 0 0 0 0 0 0 0 0 0
arr 設定值: 0 1 2 3 4 5 6 7 8 9
```

如果您想要在使用 new 新增陣列時一併指定初始值，則可以如下撰寫，注意這個方式不必指定陣列長度：

```
int[] score = new int[] {90, 85, 55, 94, 77};
```

事實上這個宣告方式是範例 5.1 中 score 陣列宣告的完整形式，範例 5.3 的是改寫範例 5.1 中 score 告方式之後的結果，執行結果與範例 5.1 則是相同的。

範例 5.3 SimpleArray2.java

```

public class SimpleArray2 {
    public static void main(String[] args) {
        int[] score = new int[] {90, 85, 55, 94, 77};

        for(int i = 0; i < score.length; i++)
            System.out.printf("score[%d] = %d\n", i, score[i]);
    }
}

```

由於陣列的記憶體空間是使用 new 配置而來，這意味著您也可以使用動態的方式來宣告陣列長度，而不用在程式中事先決定陣列大小，範例 5.4 示範了如何由使用者的輸入來決定陣列長度，它是一個計算輸入分數平均的程式。

範例 5.4 AverageInput.java

```

import java.util.Scanner;

public class AverageInput {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("請輸入學生人數: ");

        int length = scanner.nextInt();
        float[] score = new float[length]; // 動態配置長度

        for(int i = 0; i < score.length; i++) {
            System.out.print("輸入分數: ");
            float input = scanner.nextFloat();
            score[i] = input;
        }

        System.out.print("\n分數: ");
        float total = 0;
        for(int i = 0; i < score.length; i++) {
            total = total + score[i];
            System.out.print(score[i] + " ");
        }

        System.out.printf("\n平均 : %.2f", total / score.length);
    }
}

```

執行結果：

```

請輸入學生人數: 3
輸入分數: 88.3
輸入分數: 76.2
輸入分數: 90.0

分數: 88.3 76.2 90.0
平均: 84.83

```

在範例 5.4 中，您先宣告一個陣列參考名稱 score，使用 float[] score 表示 score 名稱將參考至一個元素為 float 的一維陣列物件，在使用者輸入指定長度後，您使用這個長度來配置陣列物件，並將這個陣列物件指定給 score 名稱來參考。

良葛格的話匣子 您也可以使用像是 int arr[] 這樣的方式來宣告陣列，這種宣告方式源於 C/C++ 中對陣列宣告的語法，不過在 Java 中建議使用 int[] arr 這樣的宣告方式，這也表明了 arr 是個 int[] 型態的參考名稱。

陣列的索引值由 0 開始並不是沒有原因的，事實上索引值表示的是：所指定的陣列元素相對於陣列第一個元素記憶體位置的位移量（Offset）。索引為 0 表示位移量為 0，所以就是指第一個元素，而索引 9 就是指相對於第一個元素的位移量為 9。不過在 Java 中您不直接處理關於記憶體位址的操作，以上的觀念主要是讓您更瞭解一下陣列索引的運作原理。

5.1.2 二維陣列物件

一維陣列使用「名稱」與「一個索引」來指定存取陣列中的元素，您也可以宣告二維陣列，二維陣列使用「名稱」與「兩個索引」來指定存取陣列中的元素，其宣告方式與一維陣列類似：

```
int[][] arr = {{1, 2, 3},  
               {4, 5, 6}};
```

從上面這個程式片段來看，就可以清楚的看出二維陣列的索引方式，您宣告了 2 列（Row）3 行（Column）的陣列，使用 {} 與適當的斷行可以協助您指定陣列初值，範例 5.5 簡單的示範二維陣列的存取。

範例 5.5 TwoDimArray.java

```
public class TwoDimArray {  
    public static void main(String[] args) {  
        int[][] arr = {{1, 2, 3},  
                       {4, 5, 6}};  
  
        for(int i = 0; i < arr.length; i++) {  
            for(int j = 0; j < arr[0].length; j++)  
                System.out.print(arr[i][j] + " ");  
            System.out.println();  
        }  
    }  
}
```

執行結果：

```
1 2 3  
4 5 6
```

陣列值 $arr[i][j]$ 表示指定的是第 i 列第 j 行的值。在使用二維陣列物件時，注意 $length$ 所代表的長度，陣列名稱後直接加上 $length$ （如 $arr.length$ ），所指的是有幾列（Row）；指定索引後加上 $length$ （如 $arr[0].length$ ），指的是該列所擁有的元素，也就是行（Column）數目。

良葛格的話匣子 初學者對於二維陣列的瞭解到這邊就可以了，接下來的內容是進階的二維陣列說明，初學者可以先試著瞭解，如果覺得觀念上有點難，可以先跳過待以後物件觀念更清楚後，再來看接下來的內容。

要瞭解範例 5.5 中 $length$ 成員各代表哪一個長度，您必須從物件配置的角度來瞭解。以物件的方式來配置一個二維陣列物件，您要使用以下的語法：

```
int[][] arr = new int[2][3];
```

上面這個程式片段中，您配置了 2 列 3 行的二維陣列物件，由於陣列元素的資料型態是 int ，所以陣列元素的預設元素為 0。

來細究一下二維陣列的配置細節，其實 $arr[0]$ 、 $arr[1]$ 是參考名稱，分別參考至兩個 $int[]$ 型態的物件，其長度各為 3，而 arr 名稱的型態是 $int[][],$ 參考至 $int[][],$ 型態的物件，物件中包括 $arr[0]$ 與 $arr[1]$ 兩個名稱，其關係如圖 5.1 所示。

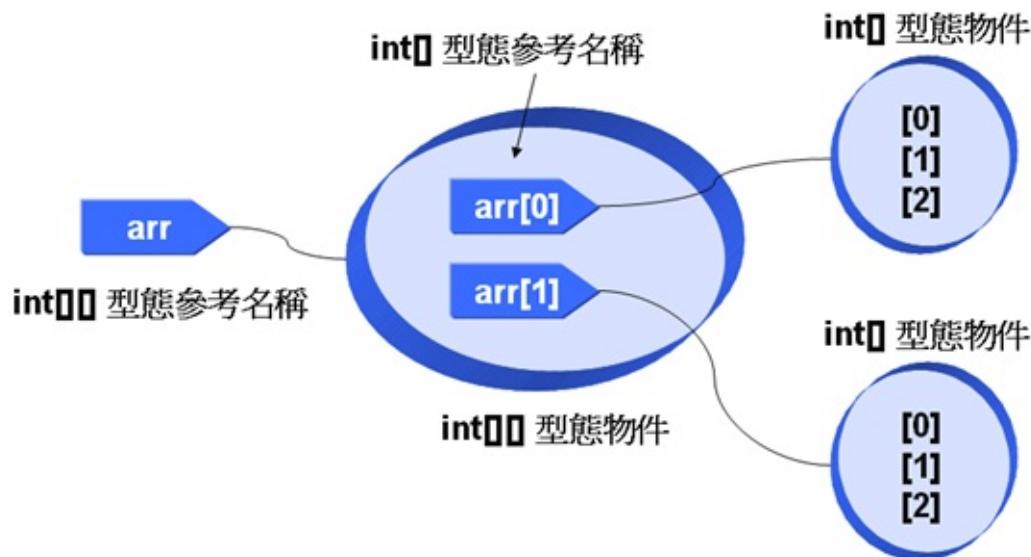


圖 5.1 二維陣列的配置關係

從圖 5.1 中您可以看到，`arr` 參考至 `int[]` 形態的物件，而 `arr[0]` 與 `arr[1]` 再分別參考至一個 `int[]` 物件，所以範例 5.5 中，使用的 `arr.length` 得到的是 2，而 `arr[0].length` 得到的長度是 3。有了陣列配置的觀念，您可以改寫範例 5.5 為範例 5.6，讓程式中的 `int[]` 型態之名稱 `foo` 來循序取出 `arr[0]` 與 `arr[1]` 所參考的 `int[]` 物件中的每個元素值，執行結果是相同的。

範例 5.6 TwoDimArray2.java

```
public class TwoDimArray2 {
    public static void main(String[] args) {
        int[][] arr = {{1, 2, 3},
                      {4, 5, 6};

        int[] foo = arr[0]; // 將arr[0] 所參考的陣列物件指定給foo

        for(int i = 0; i < foo.length; i++) {
            System.out.print(foo[i] + " ");
        }
        System.out.println();

        foo = arr[1]; // 將arr[1] 所參考的陣列物件指定給foo

        for(int i = 0; i < foo.length; i++) {
            System.out.print(foo[i] + " ");
        }
        System.out.println();
    }
}
```

如果在使用 `new` 配置二維陣列後想要一併指定初值，則可以如下撰寫：

```
int[][] arr = new int[][] {{1, 2, 3},
                           {4, 5, 6}};
```

同樣的道理，您也可以宣告三維以上的陣列，如果要同時宣告初始元素值，可以使用以下的簡便語法：

```
int[][][] arr = {
    {{1, 2, 3}, {4, 5, 6}},
    {{7, 8, 9}, {10, 11, 12}}
};
```

上面這個程式片段所宣告的三維陣列是 $2 \times 2 \times 3$ ，您將之想為兩面 2×3 二維陣列交疊在一起就是了，每一面的元素如圖 5.2 所示。



圖 5.2 三維陣列的配置關係

如果要動態宣告三維陣列，就使用以下的語法：

```
int[][][] arr = new int[2][2][3];
```

比三維以上的更多維陣列之宣告，在Java中也是可行的，但並不建議使用，使用多維陣列會讓元素索引的指定更加困難，此時適當的將資料加以分割，或是使用其它的資料結構來解決，會比直接宣告多維陣列來得實在。

由以上的說明，接下來討論「不規則陣列」。陣列的維度不一定要是四四方方的，您也可以製作一個二維陣列，而每個維度的長度並不相同，範例 5.7 是個簡單的示範。

範例 5.7 TwoDimArray3.java

```
public class TwoDimArray3 {
    public static void main(String[] args) {
        int arr[][];

        arr = new int[2][];
        arr[0] = new int[3]; // arr[0] 參考至長度為3的一維陣列
        arr[1] = new int[5]; // arr[1] 參考至長度為5的一維陣列

        for(int i = 0; i < arr.length; i++) {
            for(int j = 0; j < arr[i].length; j++)
                arr[i][j] = j + 1;
        }

        for(int i = 0; i < arr.length; i++) {
            for(int j = 0; j < arr[i].length; j++)
                System.out.print(arr[i][j] + " ");
            System.out.println();
        }
    }
}
```

這個例子只是先前說明之觀念延伸，在這個例子中，陣列第一列的長度是3，而第二列的長度是5，執行結果如下：

```
1 2 3
1 2 3 4 5
```

良葛格的話匣子 在宣告二維陣列時，也可以使用 `int arr[]` 這樣的宣告方式，這種宣告方式源於C/C++中對陣列宣告的語法，不過在 Java 中建議使用 `int[] arr` 這樣的宣告方式，這也表示了 `arr` 是個 `int[]` 型態的參考名稱；同樣的，您也

可以使用 `int arr[][][]` 這樣的方式來宣告三維陣列，但鼓勵您使用 `int arr[3][4][5]` 的宣告方式。

5.2 進階陣列觀念

陣列本身若作為物件來操作的話，會有許多特性值得討論，這個小節中將討論一些 Java 中更進階的陣列觀念，並且我也將介紹 J2SE 5.0 中對 Arrays 類別所作的功能加強（JDK6 對於 Arrays 的加強，請查看第 21 章），以及如何使用 J2SE 5.0 新增的 foreach 語法來更簡便的循序存取陣列元素。

5.2.1 進階的陣列操作

藉由對陣列物件的進一步探討，您可以稍微瞭解 Java 對物件處理的一些作法，首先來看看一維陣列的參考名稱之宣告：

```
int[] arr = null;
```

在這個宣告中，arr 表示一個可以參考至 int 一維陣列物件的參考名稱，但是目前您將這個名稱參考至 null，表示這個名稱參考還沒有參考至實際的物件，在 Java 中，'=' 運算用於基本資料型態時，是將值複製給變數，但當它用於物件時，則是將物件指定給參考名稱來參考，您也可以將同一個物件指定給兩個參考名稱，當物件的值藉由其中一個參考名稱進行操作而變更時，另一個參考名稱所參考到的值也會更動，來看看範例 5.8 的示範。

範例 5.8 AdvancedArray.java

```
public class AdvancedArray {
    public static void main(String[] args) {
        int[] arr1 = {1, 2, 3, 4, 5};
        int[] tmp1 = arr1;
        int[] tmp2 = arr1;

        System.out.print("透過tmp1取出陣列值：" );
        for(int i = 0; i < tmp1.length; i++)
            System.out.print(tmp1[i] + " ");

        System.out.print("\n透過tmp2取出陣列值：" );
        for(int i = 0; i < tmp2.length; i++)
            System.out.print(tmp2[i] + " ");

        tmp1[2] = 9;
        System.out.print("\n\n透過tmp1取出陣列值：" );
        for(int i = 0; i < tmp1.length; i++)
            System.out.print(tmp1[i] + " ");

        System.out.print("\n透過tmp2取出陣列值：" );
        for(int i = 0; i < tmp2.length; i++)
            System.out.print(tmp2[i] + " ");
        System.out.println();
    }
}
```

執行結果：

```
透過tmp1取出陣列值：1 2 3 4 5
透過tmp2取出陣列值：1 2 3 4 5

透過tmp1取出陣列值：1 2 9 4 5
透過tmp2取出陣列值：1 2 9 4 5
```

在這個範例中，您藉由 tmp1 名稱改變了索引 2 的元素值，由於 tmp2 也參考至同一陣列物件，所以 tmp2 取出索引 2 的元素值是改變後的值，事實上在範例 5.8 中，有三個參考名稱參考至同一個陣列物件，也就是 arr1、tmp1 與 tmp2，如下圖所示：

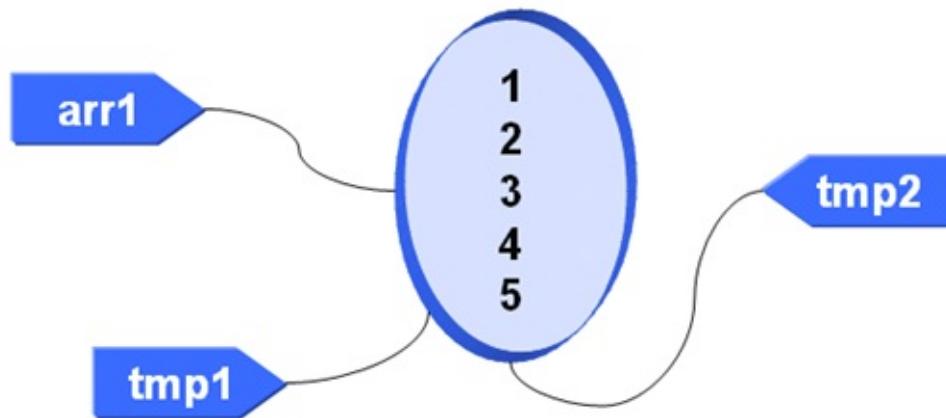


圖 5.3 三個名稱參考至同一物件

所以您應該知道，如果取出 arr1 索引 2 的元素，元素值也會是 9。

在宣告 int[] arr 之後，arr 是一個一維陣列物件的參考名稱，所以它可以參考至任何長度的一維陣列物件，這邊使用範例 5.9 來作示範。

範例 5.9 AdvancedArray2.java

```
public class AdvancedArray2 {
    public static void main(String[] args) {
        int[] arr1 = {1, 2, 3, 4, 5};
        int[] arr2 = {5, 6, 7};
        int[] tmp = arr1;

        System.out.print("使用tmp取出arr1中的元素：");
        for(int i = 0; i < tmp.length; i++)
            System.out.print(tmp[i] + " ");

        tmp = arr2;
        System.out.print("\n使用tmp取出arr2中的元素：");
        for(int i = 0; i < tmp.length; i++)
            System.out.print(tmp[i] + " ");
        System.out.println();
    }
}
```

在範例 5.9 中，tmp 可以參考至擁有 5 個元素的一維陣列，也可以參考至擁有 3 個元素的一維陣列，執行結果如下：

```
使用tmp取出arr1中的元素：1 2 3 4 5
使用tmp取出arr2中的元素：5 6 7
```

您瞭解到在 Java 中陣列是一個物件，而使用 '=' 指定時是將物件指定給陣列名稱來參考，也就是相當於圖 5.3 中改變名稱所綁定的物件，而不是將陣列進行複製，如果您想將整個陣列的值複製給另一個陣列該如何作呢？您可以使用迴圈，將整個陣列的元素值走訪一遍，並指定給另一個陣列相對應的索引位置，範例 5.10 示範了進行陣列複製的方法。

範例 5.10 ArrayCopy.java

```
public class ArrayCopy {
    public static void main(String[] args) {
        int[] arr1 = {1, 2, 3, 4, 5};
        int[] arr2 = new int[5];

        for(int i = 0; i < arr1.length; i++)
            arr2[i] = arr1[i];
```

```

    for(int i = 0; i < arr2.length; i++)
        System.out.print(arr2[i] + " ");
    System.out.println();
}
}

```

執行結果：

```
1 2 3 4 5
```

另一個進行陣列複製的方法是使用 `System` 類別所提供的 `arraycopy()` 方法，其語法如下：

```
System.arraycopy(來源, 起始索引, 目的, 起始索引, 複製長度);
```

範例 5.11 改寫了範例 5.10，使用 `System.arraycopy()` 進行陣列複製，執行結果與範例 5.10 是相同的。

範例 5.11 ArrayCopy2.java

```

public class ArrayCopy2 {
    public static void main(String[] args) {
        int[] arr1 = {1, 2, 3, 4, 5};
        int[] arr2 = new int[5];

        System.arraycopy(arr1, 0, arr2, 0, arr1.length);

        for(int i = 0; i < arr2.length; i++)
            System.out.print(arr2[i] + " ");
        System.out.println();
    }
}

```

在 JDK6 中，也為 `Arrays` 類別新增了陣列複製的 `copyOf()` 方法，詳情請查看第 21 章。

5.2.2 Arrays 類別

對陣列的一些基本操作，像是排序、搜尋與比較等動作是很常見的，在 Java 中提供了 `Arrays` 類別可以協助您作這幾個動作，`Arrays` 類別位於 `java.util` 套件中，它提供了幾個方法可以直接呼叫使用。

表 5.2 Arrays 類別提供的幾個方法說明

名稱	說明
<code>sort()</code>	幫助您對指定的陣列排序，所使用的是快速排序法
<code>binarySearch()</code>	讓您對已排序的陣列進行二元搜尋，如果找到指定的值就傳回該值所在的索引，否則就傳回負值
<code>fill()</code>	當您配置一個陣列之後，會依資料型態來給定預設值，例如整數陣列就初始為 0，您可以使用 <code>Arrays.fill()</code> 方法來將所有的元素設定為指定的值
<code>equals()</code>	比較兩個陣列中的元素值是否全部相等，如果是將傳回 <code>true</code> ，否則傳回 <code>false</code>

範例 5.12 示範了使用 `Arrays` 來進行陣列的排序與搜尋。

範例 5.12 ArraysMethodDemo.java

```

import java.util.Scanner;
import java.util.Arrays;

public class ArraysMethodDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int[] arr = {93, 5, 3, 55, 57, 7, 2, 73, 41, 91};

        System.out.print("排序前: ");
        for(int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();

        Arrays.sort(arr);

        System.out.print("排序後: ");
        for(int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }

        System.out.print("\n請輸入搜尋值: ");
        int key = scanner.nextInt();
        int find = -1;
        if((find = Arrays.binarySearch(arr, key)) > -1) {
            System.out.println("找到值於索引 " +
                find + " 位置");
        } else {
            System.out.println("找不到指定值");
        }
    }
}

```

執行結果：

```

排序前: 93 5 3 55 57 7 2 73 41 91
排序後: 2 3 5 7 41 55 57 73 91 93
請輸入搜尋值: 7
找到值於索引 3 位置

```

範例 5.13 示範了使用 Arrays 來進行陣列的填充與比較。

範例 5.13 ArraysMethodDemo2.java

```

import java.util.Arrays;

public class ArraysMethodDemo2 {
    public static void main(String[] args) {
        int[] arr1 = new int[10];
        int[] arr2 = new int[10];
        int[] arr3 = new int[10];

        Arrays.fill(arr1, 5);
        Arrays.fill(arr2, 5);
        Arrays.fill(arr3, 10);

        System.out.print("arr1: ");
        for(int i = 0; i < arr1.length; i++) {
            System.out.print(arr1[i] + " ");
        }

        System.out.println("\narr1 = arr2 ? " +
            Arrays.equals(arr1, arr2));
        System.out.println("arr1 = arr3 ? " +
            Arrays.equals(arr1, arr3));
    }
}

```

執行結果：

```
arr1: 5 5 5 5 5 5 5 5 5
arr1 = arr2 ? true
arr1 = arr3 ? false
```

請注意到，您不可以使用 '`==`' 來比較兩個陣列的元素值是否相等，'`==`' 使用於物件比對時，是用來測試兩個物件名稱是否參考至同一個物件，也就是測試兩個名稱是不是綁定至同一個物件，範例 5.14 是這個觀念的實際示範。

範例 5.14 TestArrayValue.java

```
public class TestArrayValue {
    public static void main(String[] args) {
        int[] arr1 = {1, 2, 3, 4, 5};
        int[] arr2 = {1, 2, 3, 4, 5};

        int[] tmp = arr1;

        System.out.println(arr1 == tmp);
        System.out.println(arr2 == tmp);
    }
}
```

在範例 5.14 中，雖然 `arr1` 與 `arr2` 中的元素值是相同的，但實際上 `arr1` 與 `arr2` 是參考至不同的兩個陣列物件，您將 `arr1` 指定給 `tmp` 來參考，由於 `tmp` 與 `arr1` 是參考同一陣列物件，如圖 5.4 所示：

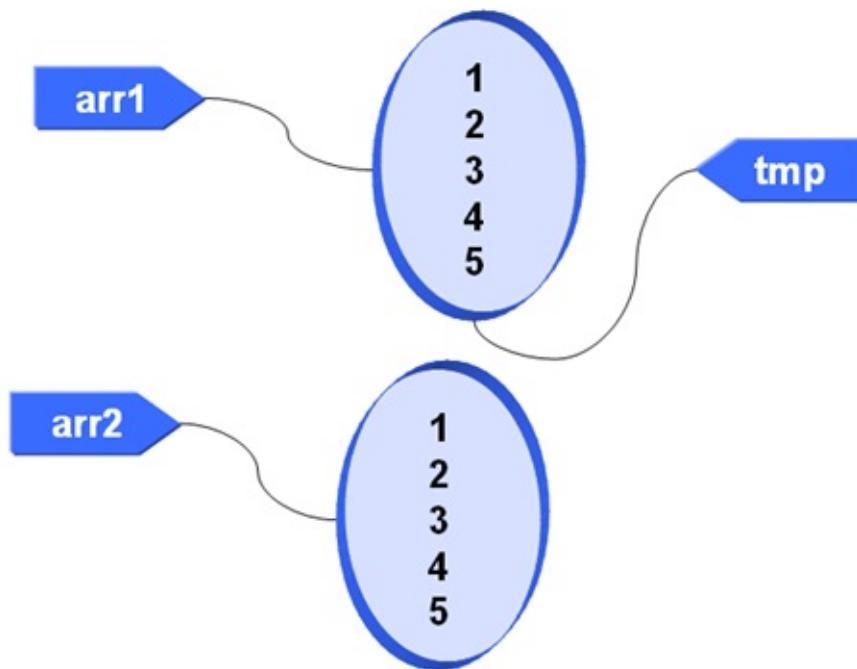


圖 5.4 `arr1` 與 `tmp` 是參考至同一物件

所以進行 '`arr1==tmp`' 比較時會顯示 `true`，而 `tmp` 與 `arr2` 是參考至不同陣列物件，所以進行 '`arr2==tmp`' 比較時會顯示 `false`，執行結果如下：

```
true
false
```

在 J2SE 5.0 中對 `Arrays` 類別作了不少的修改與功能新增，由此可見陣列操作在程式中的重要性，這邊介紹 `Arrays` 中新增的兩個方法：`deepEquals()` 與 `deepToString()`。

表 5.3 Arrays 類別新增的兩個方法說明

名稱	說明
deepEquals()	對陣列作深層比較，簡單的說，您可以對二維乃至三維以上的陣列進行比較是否相等
deepToString()	將陣列值作深層輸出，簡單的說，您可以對二維乃至三維以上的陣列輸出其字串值

範例 5.15 是個簡單示範，它對三個二維陣列進行深層比較與深層輸出。

範例 5.15 NewArraysDemo.java

```
import java.util.Arrays;

public class NewArraysDemo {
    public static void main(String args[]) {
        int[][] arr1 = {{1, 2, 3},
                        {4, 5, 6},
                        {7, 8, 9}};
        int[][] arr2 = {{1, 2, 3},
                        {4, 5, 6},
                        {7, 8, 9}};
        int[][] arr3 = {{0, 1, 3},
                        {4, 6, 4},
                        {7, 8, 9}};

        System.out.println("arr1 內容等於 arr2 ? " +
                           Arrays.deepEquals(arr1, arr2));
        System.out.println("arr1 內容等於 arr3 ? " +
                           Arrays.deepEquals(arr1, arr3));
        System.out.println("arr1 deepToString()\n" +
                           Arrays.deepToString(arr1));
    }
}
```

執行結果：

```
arr1 內容等於 arr2 ? true
arr1 內容等於 arr3 ? false
arr1 deepToString()
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

當然 Arrays 並不只有以上介紹的功能，總之，如果您之前對 Arrays 沒這麼的重視，現在您可以多關照它幾眼，如果您有陣列操作方面的相關需求，可以先查查 `java.util.Arrays` 的 API 文件說明，看看有沒有現成的方法可以使用。

5.2.3 foreach 與陣列

J2SE 5.0 新增了 `foreach` 的語法，又稱加強的 `for` 迴圈（Enhanced for Loop），其應用的對象之一是在陣列的循序存取上，`foreach` 語法如下：

```
for(type element : array) {
    System.out.println(element)...
}
```

直接以實例的方式來說明會更容易瞭解這個語法如何使用，在 J2SE 5.0 之前您可以使用以下的方式來循序存取陣列中的元素：

```
int[] arr = {1, 2, 3, 4, 5};
for(int i = 0; i < arr.length; i++)
```

```
System.out.println(arr[i]);
```

在 J2SE 5.0 中可以使用新的 foreach 語法這麼寫：

```
int[] arr = {1, 2, 3, 4, 5};
for(int element : arr)
    System.out.println(element);
```

每一次從 arr 中取出的元素，會自動設定給 element，您不用自行判斷是否超出了陣列的長度，注意 element 的型態必須與陣列元素的元素的型態相同。

如果是物件的話，作法也是類似，例如存取字串陣列的話，可以如下撰寫：

```
String[] names = {"caterpillar", "momor", "bush"};
for(String name : names)
    System.out.println(name);
```

那麼二維陣列呢？基本上您要是瞭解陣列本身就是個物件，您自然就會知道如何存取，舉個例子：

```
int[][] arr = {{1, 2, 3},
               {4, 5, 6},
               {7, 8, 9}};
for(int[] row : arr) {
    for(int element : row) {
        System.out.println(element);
    }
}
```

三維以上的陣列使用 foreach 的方式來存取也可以依此類推。

5.2.4 物件陣列

如果使用類別型態來宣告陣列，有幾個常見的有趣問題，在這邊先一步一步來看，首先請問您，以下產生幾個物件：

```
int[] arr = new int[3];
```

這個是很基本的問題，答案是一個一維陣列物件，由於元素型態是 int，所以每個元素值初始值是 0。那麼以下的宣告產生幾個物件：

```
int[][] arr = new int[2][3];
```

答案是 3 個，理由可以看看先前圖 5.1 的圖解就可以明白。現在再請問，以下產生幾個物件：

```
Integer[] arr = new Integer[3];
```

有的人會以為這樣會產生 3 個 Integer 的實例，但事實上不是，以上產生的是 1 個一維陣列，由於元素型態是 Integer，所以元素值全部參考至 null，如圖 5.5 所示：

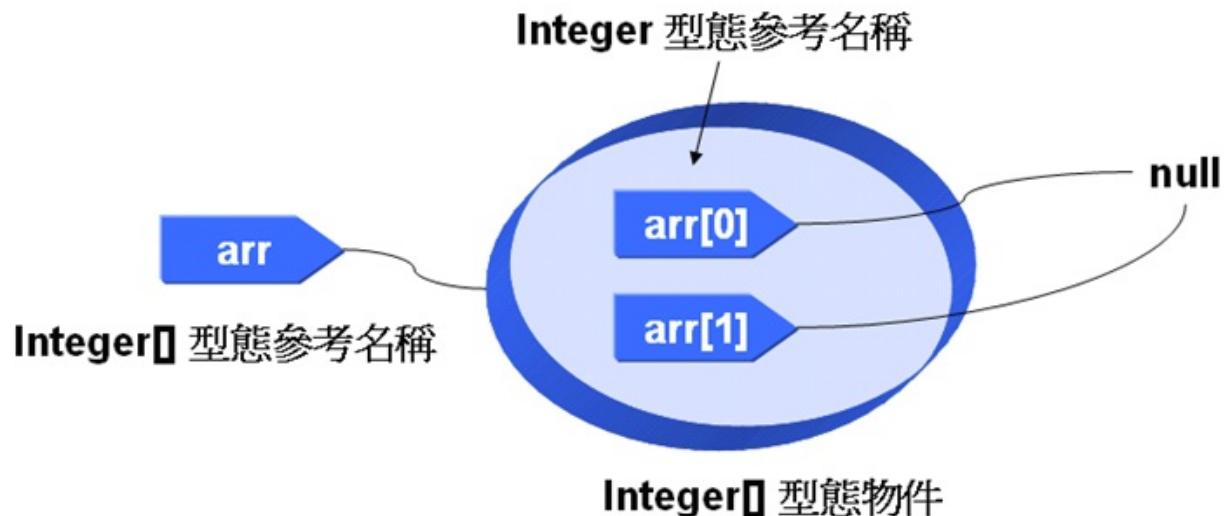


圖 5.5 一維物件陣列示意圖

最後一個問題，請問以下的宣告產生幾個物件？

```
Integer[][] arr = new Integer[2][3];
```

不好想像嗎？這時畫個圖就很清楚了，如圖 5.6 所示：

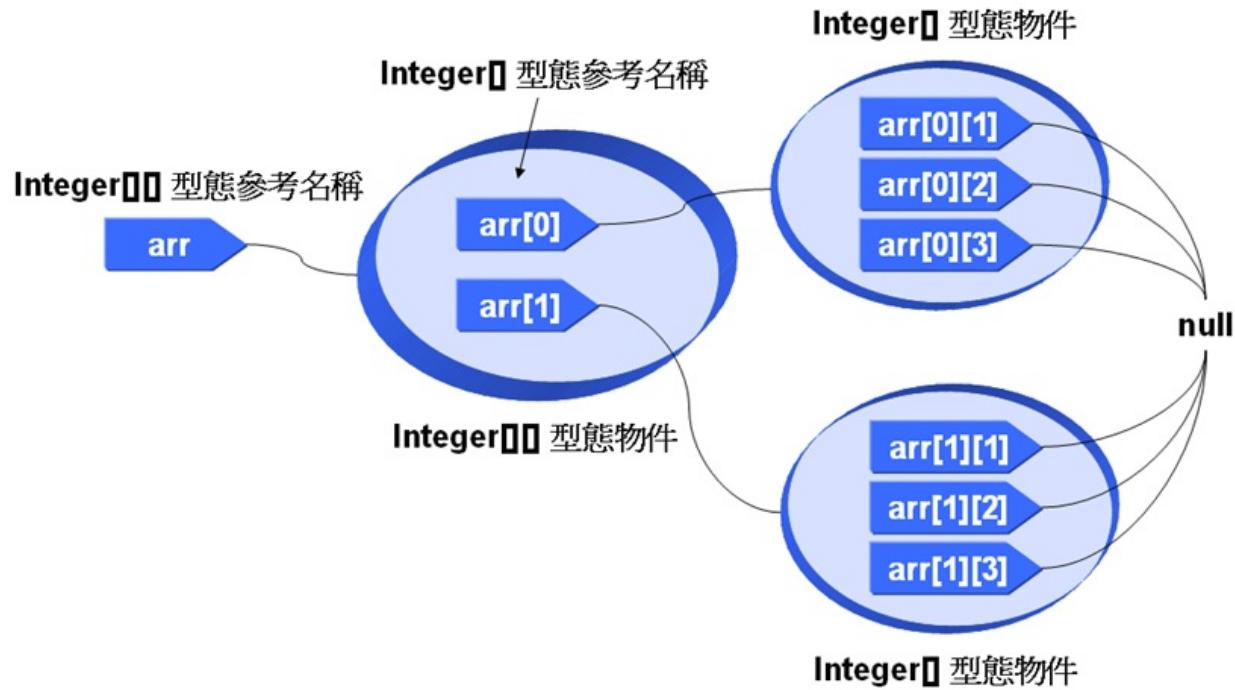


圖 5.6 二維物件陣列示意圖

當您搞不清楚物件之間的配置關係時，畫圖是很好的表示方式，在上圖中，我們可以看到有3個物件，而由於元素型態是 Integer，所以六個元素參考名稱預設都是參考至 null。

5.3 接下來的主題

每一個章節的內容由淺至深，初學者該掌握的深度要到哪呢？在這個章節中，對於初學者我建議至少掌握以下幾點內容：

- 會宣告一維、二維陣列
- 會循序取出一維、二維陣列的元素並顯示在文字模式下
- 記得表 5.1 的陣列元素初始值
- 會使用 Arrays 類別來協助您進行陣列操作

接下來要瞭解的字串（String）與陣列有一些相類似的特性，字串的本質是字元陣列，但在 Java 中字串不僅是字元陣列，而是 String 類別的一個實例，在 Java 中您可以用一些簡單的方式處理字串，不過有些字串的特性是您必須瞭解的，這可以讓您的程式在處理字串上更有效率。

第 6 章 字串

字串也許是您在 Java 中最常處理的物件，但也可能是最常被忽視的物件，從這個章節開始，無論如何請重視它，字串運用的好不好，足以影響您程式運行的效率，瞭解字串上可用的操作，則可以讓您省去不少程式撰寫與維護的功夫。

這個章節會將字串（String）分作兩個部份來討論：一個是字串本身的特性，您要瞭解這些特性，才知道如何對字串進行比較、如何將字串用的有效率；一個是介紹字串上可用的幾個操作與可用的 API 輔助，像是字串的分離、正則式（Regular expression）比對、Pattern 與 Matcher 的使用等，您也可以學到如何直接在啟動程式的同時下引數給程式來使用。

6.1 認識字串

字串的本質是字元（char）型態的陣列，在 Java 中則更將字串視為 String 類別的一個實例，也就是將其視為物件存在於程式之中，這讓字串處理在 Java 中變得簡單，這個小節就先從基本的字串特性開始認識。

6.1.1 String 類別

由字元所組成的一串文字符號被稱之為「字串」，例如 "Hello" 這個字串就是由 'H'、'e'、'l'、'l'、'o' 這五個字元所組成，在某些程式語言中，字串是以字元陣列的方式存在：

“Hello”



圖 6.1 字串的本質是字元陣列

在 Java 中字串不僅僅是字元陣列，而是 String 類別的一個實例，可以使用 String 類別來建構，例如您可以使用以下的方式來宣告一個字串參考名稱，並指向一個字串實例：

```
String text = "字串的使用";
System.out.println(text);
```

注意字串的直接指定必須使用 "" 來包括您的文字，字串的每個字元是使用 Unicode 字元來建構，在建構一個字串物件之後，您可以直接在輸出串流（out）中指定字串物件的參考名稱來輸出字串。

字串的串接在 Java 中可以直接使用 '+' 運算子，'+' 本來是加法運算子，而它被重新定義（Override）為可以直接用於字串的串接，例如您可以如下將兩個字串接在一起輸出：

```
String msg = "哈囉！";
msg = msg + "Java 程式設計！";
System.out.println(msg);
```

這一段程式碼會在文字模式上顯示 "哈囉！Java 程式設計！"。字串在 Java 中以 String 類別的一個實例存在，所以每個字串物件本身會擁有幾個可操作的方法，表 6.1 先介紹幾個常用的方法。

表 6.1 String 物件上的幾個方法

方法	說明
length()	取得字串的字元長度
equals()	判斷原字串中的字元是否相等於指定字串中的字元
toLowerCase()	轉換字串中的英文字元為小寫

toUpperCase()

轉換字串中的英文字元為大寫

範例 6.1 示範了以上的幾個字串操作方法的使用。

範例 6.1 StringDemo.java

```
public class StringDemo {
    public static void main(String[] args) {
        String text = "hello";

        System.out.println("字串內容: " + text);
        System.out.println("字串長度: " + text.length());
        System.out.println("等於hello? " +
                           text.equals("hello"));
        System.out.println("轉為大寫: " +
                           text.toUpperCase());
        System.out.println("轉為小寫: " +
                           text.toLowerCase());
    }
}
```

執行結果：

```
字串內容: hello
字串長度: 5
等於hello? true
轉為大寫: HELLO
轉為小寫: hello
```

如果您要將輸入的字串轉換為整數、浮點數等資料型態，您可以使用表 6.2 中各類別所提供的各個靜態（static）剖析方法。

表 6.2 將字串剖析為數值型態

方法	說明
Byte.parseByte(字串)	將字串剖析為位元
Short.parseShort(字串)	將字串剖析為short整數
Integer.parseInt(字串)	將字串剖析為int整數
Long.parseLong(字串)	將字串剖析為long整數
Float.parseFloat(字串)	將字串剖析為float浮點數
Double.parseDouble(字串)	將字串剖析為double浮點數

注意如果指定的字串無法剖析為指定的資料型態數值，則會發生 NumberFormatException 例外。之前宣告字串時，都是以這樣的樣子來宣告：

```
String str = "caterpillar";
```

這樣的宣告方式看來像是基本資料型態宣告，但事實上 String 並不是 Java 的基本資料型態，String 是 java.lang 套件下所提供的類別，如果以配置物件的觀念來宣告字串，應該是這樣的：

```
String str = new String("caterpillar");
```

不過事實上兩種宣告方式是有所差別的，這待會 6.1.2 再詳細說明，在這邊只要先記得這個差異性就可以了。

字串的本質是由字元陣列所組成，所以使用 String 類別宣告字串後，該字串會具有陣列索引的性質，表 6.3 介紹幾個與使用索引取得字元的相關方法。

表 6.3 取得字串中的字元之方法

方法	說明
char charAt(int index)	傳回指定索引處的字元
int indexOf(int ch)	傳回指定字元第一個找到的索引位置
int indexOf(String str)	傳回指定字串第一個找到的索引位置
int lastIndexOf(int ch)	傳回指定字元最後一個找到的索引位置
String substring(int beginIndex)	取出指定索引處至字串尾端的子字串
String substring(int beginIndex, int endIndex)	取出指定索引範圍子字串
char[] toCharArray()	將字串轉換為字元陣列

範例 6.2 為表 6.3 作了幾個簡單的示範。

範例 6.2 CharAtString.java

```
public class CharAtString {
    public static void main(String[] args) {
        String text = "One's left brain has nothing right.\n"
                    + "One's right brain has nothing left.\n";

        System.out.println("字串內容: ");
        for(int i = 0; i < text.length(); i++)
            System.out.print(text.charAt(i));

        System.out.println("\n第一個left: " +
                           text.indexOf("left"));
        System.out.println("最後一個left: " +
                           text.lastIndexOf("left"));

        char[] charArr = text.toCharArray();
        System.out.println("\n字元Array內容: ");
        for(int i = 0; i < charArr.length; i++)
            System.out.print(charArr[i]);
    }
}
```

執行結果：

```
字串內容:
One's left brain has nothing right.
One's right brain has nothing left.

第一個left: 6
最後一個left: 66

字元Array內容:
One's left brain has nothing right.
One's right brain has nothing left.
```

在建構字串物件時，除了直接在 '=' 後使用 "" 來指定字串常數之外，您也可以使用字元陣列來建構，例如使用字元陣列 name，建構出一個內容為 "caterpillar" 的字串：

```
char[] name = {'c', 'a', 't', 'e', 'r', 'p', 'i', 'l', 'l', 'a', 'r'};
String str = new String(name);
```

除了以上所介紹的幾個方法之外，您應該查詢 API 手冊，瞭解更多有關於 String 類別的方法，例如 String 上還有 endsWith() 方法可以判斷字串是不是以指定的文字作為結束，您可以使用這個方法來過濾檔案名稱，範例 6.3 就使用 endsWith() 來過濾出副檔名為 jpg 的檔案。

範例6.3 FileFilter.java

```
public class FileFilter {
    public static void main(String[] args) {
        String[] filenames = {"caterpillar.jpg", "cater.gif",
            "bush.jpg", "wuwu.jpg", "clockman.gif"};
        System.out.print("過濾出jpg檔案: ");
        for(int i = 0; i < filenames.length; i++) {
            if(filenames[i].endsWith("jpg")) {
                System.out.print(filenames[i] + " ");
            }
        }
        System.out.println("");
    }
}
```

執行結果：

```
過濾出jpg檔案: caterpillar.jpg bush.jpg wuwu.jpg
```

良葛格的話匣子 在宣告字串一併指定字串值時，如果字串長度過長，可以分作兩行來寫，這樣比較容易閱讀，例如：

```
String text = "One's left brain has nothing right.\n"
+ "One's right brain has nothing left.\n";
```

6.1.2 不可變 (immutable) 字串

在 Java 中使用字串有一個非常重要的觀念必須記得，一個字串物件一旦被配置，它的內容就是固定不可變的 (immutable)，例如下面這個宣告：

```
String str = "caterpillar";
```

這個宣告會配置一個長度為 11、內容為 "caterpillar" 的字串物件，您無法改變這個字串物件的內容；不要以為下面的陳述就是改變一個字串物件的內容：

```
String str = "Just";
str = "Justin";
```

事實上在這個程式片段中會有兩個字串物件，一個是 "Just" 字串物件，長度為 4，一個是 "Justin" 字串物件，長度為 6，兩個是不同的字串物件，您並不是在 "Just" 字串後加上 "in" 字串，而是讓 str 名稱參考至 "Justin" 字串物件，圖 6.2 展示出這個概念。

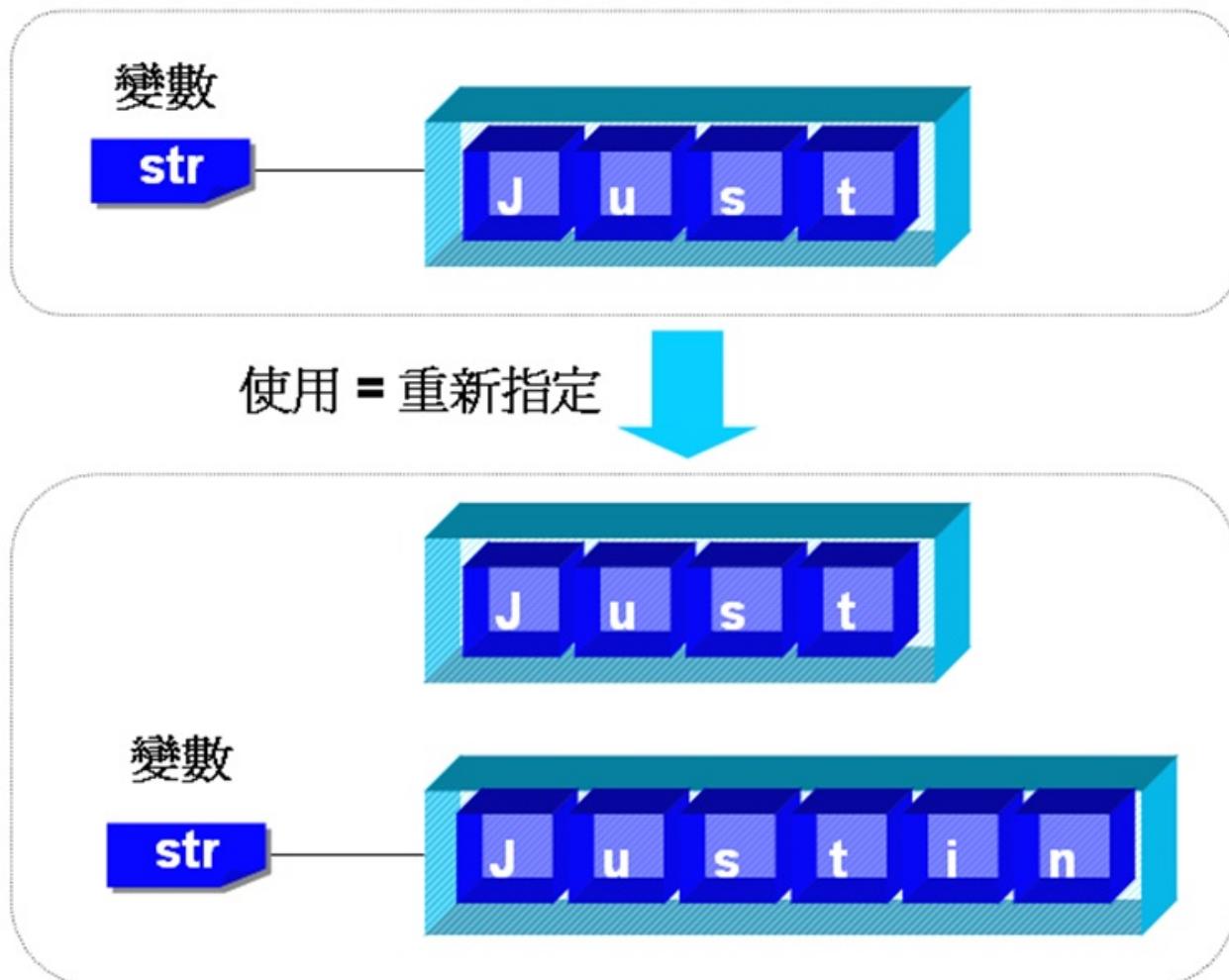


圖 6.2 使用 '=' 作字串指定的意義

在 Java 中，使用 '=' 將一個字串物件指定給一個參考名稱，其意義為改變該名稱所參考的物件，原來被參考的字串物件若沒有其它名稱來參考它，就會在適當的時機被 Java 的「垃圾回收」（Garbage collection）機制回收，在 Java 中，程式設計人員通常不用關心無用物件的資源釋放問題，Java 會檢查物件是否不再被參考，如果沒有被任何名稱參考的物件將會被回收。

在 Java 執行時會維護一個 String 池（Pool），對於一些可以共享的字串物件，會先在 String 池中查找是否存在相同的 String 內容（字元相同），如果有就直接傳回，而不是直接創造一個新的 String 物件，以減少記憶體的耗用，如果您在程式中使用下頁的方式來宣告，則實際上是指向同一個字串物件：

```
String str1 = "flyweight";
String str2 = "flyweight";
System.out.println(str1 == str2);
```

當您直接在程式中使用 "" 來包括一個字串時，該字串就會在 String 池中，上面的程式片段中的 "flyweight" 就是這樣的情況，"flyweight" 在程式中會有一個實例，而 str1 與 str2 同時參考至 "flyweight"，圖 6.3 表示了這個概念。

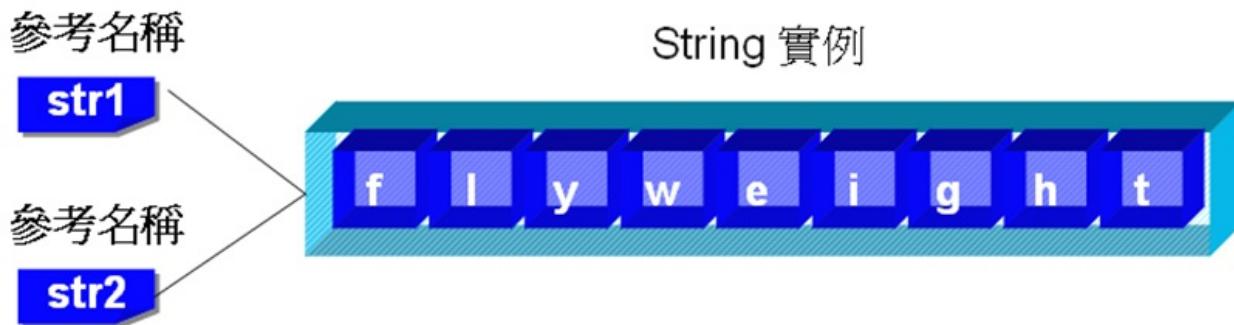


圖 6.3 字串參考名稱`str1`、`str2`同時參考至"flyweight"

在 Java 中如果 `'=='` 被使用於兩個參考名稱時，它是用於比較兩個參考名稱是否參考至同一物件，所以在圖 6.3 中當 `str1==str2` 比較時，程式的執行結果會顯示 `true`。

再來看看關於 `String` 的 `intern()` 方法，先看一下 API 說明的節錄（翻譯）：

在 `intern()` 方法被呼叫時，如果池（Pool）中已經包括了相同的 `String` 物件（相同與否由 `equals()` 方法決定），那麼會從池中返回該字串，否則的話

這段話其實說明了「Flyweight模式」（請見本章後的網路索引）的運作方式，直接使用範例 6.4 來說明會更清楚。

範例 6.4 InternString.java

```
public class InternString {
    public static void main(String[] args) {
        String str1 = "fly";
        String str2 = "weight";
        String str3 = "flyweight";
        String str4 = null;

        str4 = str1 + str2;
        System.out.println(str3 == str4);

        str4 = (str1 + str2).intern();
        System.out.println(str3 == str4);
    }
}
```

使用圖形來說明這個範例，在宣告 `str1`、`str2`、`str3` 後，`String` 池中的狀況如圖 6.4 所示。

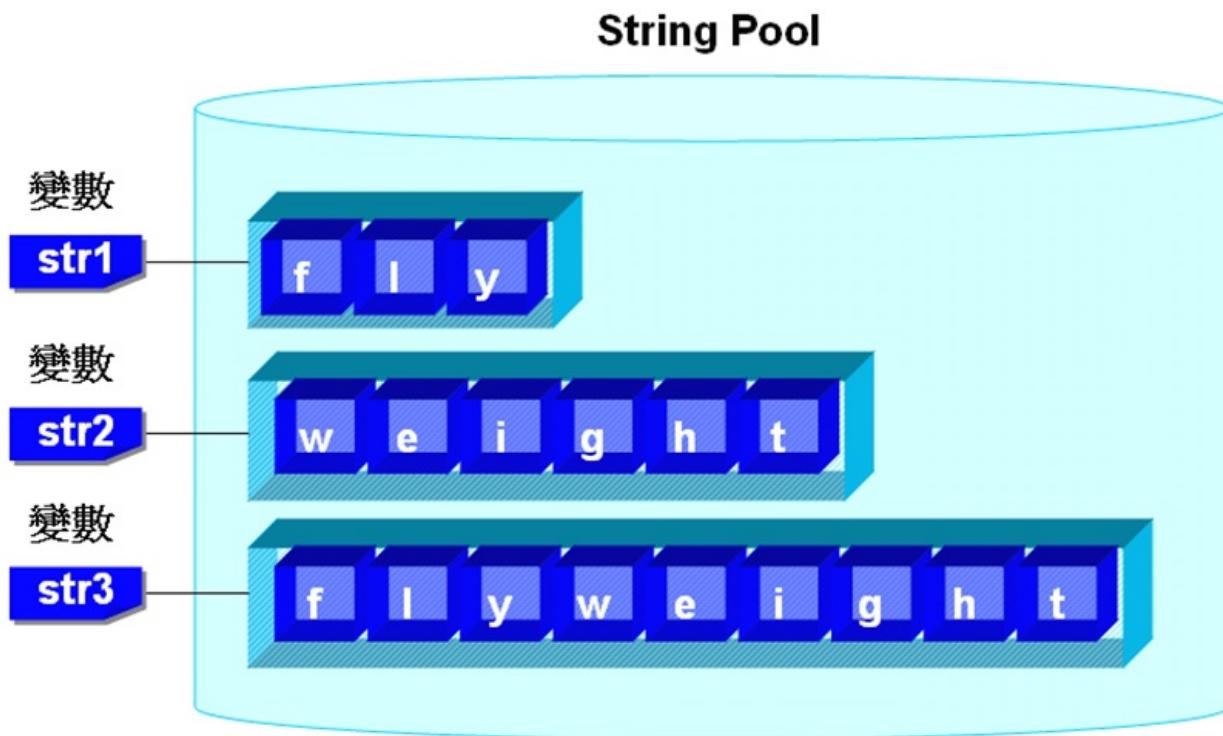


圖 6.4 字串池示意

在 Java 中，使用 '+' 串接字串的話會產生一個新的字串物件，所以在程式中第一次比較 str3 與 str4 物件是否為同一物件時，結果會是 false，如圖 6.5 所示。

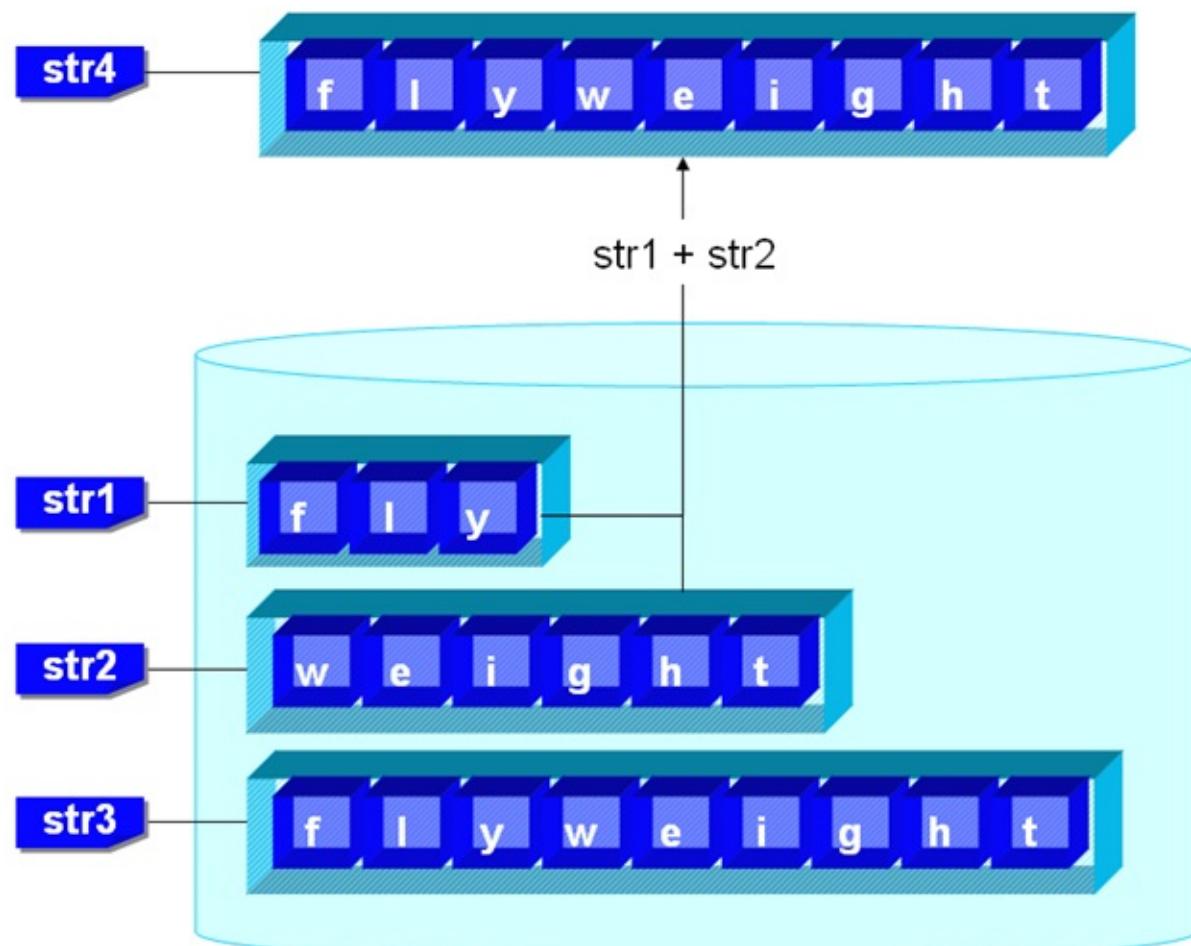
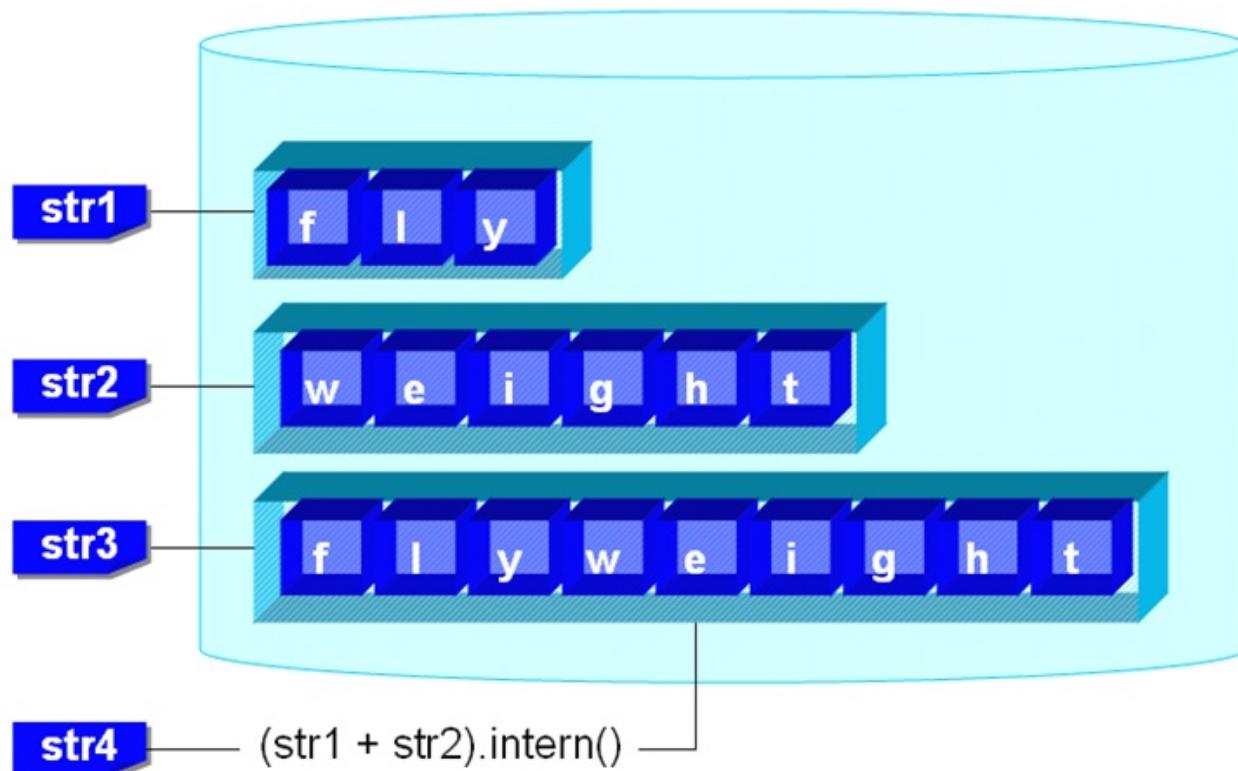


圖 6.5 字串加法會產生新的字串

`intern()` 方法會先檢查 String 池中是否存在字元部份相同的字串物件，如果有的話就傳回，由於程式中之前已經有宣告 "flyweight" 字串物件，`intern()` 在 String 池中發現了它，所以直接傳回，這時再進行比較，`str3` 與 `str4` 所指向的其實會是同一物件，所以結果會是 true，如圖 6.6 所示。

圖 6.6 `intern()` 會返回 String 池中字串物件之參考

由以上的說明也帶出一個觀念，'==' 在 Java 中被用來比較兩個參考名稱是否參考至同一物件，所以您不可以用 '==' 來比較兩個字串的字元內容是否相同，例如：

```
String str1 = new String("caterpillar");
String str2 = new String("caterpillar");
System.out.println(str1 == str2);
```

雖然兩個字串物件的字元值完全相同，但實際上在這個程式片段中，您產生了兩個 String 的實例，`str1` 與 `str2` 分別參考至不同的實例，所以使用 '==' 比較時結果會顯示 false，如果您要比較兩個字串物件的字元值是否相同，您要使用 `equals()` 方法，以下的寫法才會顯示 true 的結果：

```
String str1 = new String("caterpillar");
String str2 = new String("caterpillar");
System.out.println(str1.equals(str2));
```

一個常見的問題是：上面的程式片段產生了幾個 String 的實例？很多人會回答 2 個，但答案是 3 個，因為 "caterpillar" 就是一個，它存在於字串池中，而您又使用 `new` 建構出兩個 String 物件，分別由 `str1` 與 `str2` 參考，所以總共會有 3 個 String 實例。

6.1.3 StringBuilder 類別

一個 String 物件的長度是固定的，您不能改變它的內容，或者是附加新的字元至 String 物件中，您也許會使用 '+' 來串接字

串以達到附加新字元或字串的目的，但 '+' 會產生一個新的 String 實例，如果您的程式對這種附加字串的需求很頻繁，並不建議使用 '+' 來進行字串的串接，在物件導向程式設計中，最好是能重複運用已生成的物件，物件的生成需要記憶體空間與時間，不斷的產生 String 實例是一件沒有效率的行為。

在 J2SE 5.0 開始提供 `java.lang.StringBuilder` 類別，使用這個類別所產生的物件預設會有 16 個字元的長度，您也可以自行指定初始長度，如果附加的字元超出可容納的長度，則 `StringBuilder` 物件會自動增加長度以容納被附加的字元，如果您有頻繁作字串附加的需求，使用 `StringBuilder` 會讓程式的效率大大提昇，來寫個簡單的測試程式就可以知道效能差距有多大。

範例 6.5 AppendStringTest.java

```
public class AppendStringTest {
    public static void main(String[] args) {
        String text = "";
        long beginTime = System.currentTimeMillis();
        for(int i = 0; i < 10000; i++)
            text = text + i;
        long endTime = System.currentTimeMillis();
        System.out.println("執行時間：" + (endTime - beginTime));

        StringBuilder builder = new StringBuilder("");
        beginTime = System.currentTimeMillis();
        for(int i = 0; i < 10000; i++)
            builder.append(String.valueOf(i));
        endTime = System.currentTimeMillis();
        System.out.println("執行時間：" + (endTime - beginTime));
    }
}
```

在範例 6.5 中首先使用 '+' 來串接字串，您使用 `System.currentTimeMillis()` 取得 for 迴圈執行前、後的系統時間，如此就可以得知 for 迴圈執行了多久，以下是我的電腦上的測試數據：

```
執行時間：4641
執行時間：16
```

您可以看到執行的時間差距很大，這說明了使用 '+' 串接字串所帶來負擔，如果您有經常作附加字串的需求，建議使用 `StringBuilder`，事實上就範例 6.5 來說，第二個 for 迴圈執行時間還可以更短，因為 `append()` 也可以接受基本資料型態，所以不必特地使用 `String.valueOf()` 方法從 int 取得 String，改為以下的方式，執行時間可以更大幅縮短：

```
for(int i = 0; i < 10000; i++)
    builder.append(i);
```

使用 `StringBuilder` 最後若要輸出字串結果，可以呼叫其 `toString()` 方法，您可以使用 `length()` 方法得知目前物件中的字元長度，而 `capacity()` 可傳回該物件目前可容納的字元容量，另外 `StringBuilder` 還有像是 `insert()` 方法可以將字元插入指定的位置，如果該位置以後有字元，則將所有的字元往後移；`deleteChar()` 方法可以刪除指定位置的字元，而 `reverse()` 方法可以反轉字串，詳細的使用可以查詢看看 `java.lang.StringBuilder` 的 API 文件說明。

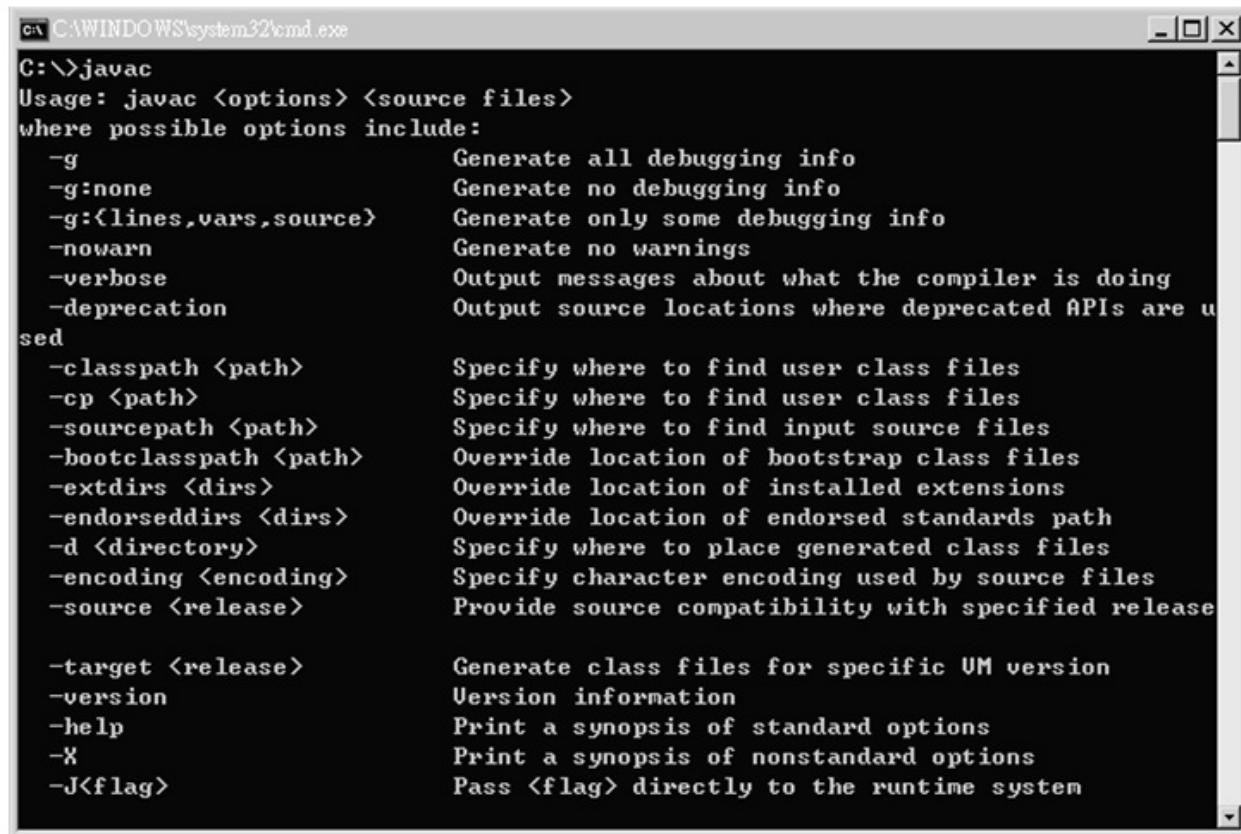
`StringBuilder` 是 J2SE 5.0 才新增的類別，在 J2SE 5.0 之前的版本若有相同的需求，是使用 `java.lang.StringBuffer`，事實上，`StringBuilder` 被設計為與 `StringBuffer` 具有相同的操作介面，在單機非「多執行緒」（Multithread）的情況下使用 `StringBuilder` 會有較好的效率，因為 `StringBuilder` 沒有處理「同步」（Synchronized）問題；`StringBuffer` 則會處理同步問題，如果您的 `StringBuilder` 會在多執行緒下被操作，則要改用 `StringBuffer`，讓物件自行管理同步問題，關於多執行緒的觀念，會在第 15 章詳細說明。

6.2 字串進階運用

在瞭解字串的基本特性之後，接下來看看如何操作字串，除了 String 類別上的幾個可用方法之外，您還可以使用一些輔助的類別，像是「正則表示式」（Regular expression）於字串比對上的應用。

6.2.1 命令列引數

在文字模式下執行程式時，通常您可以連帶指定一些引數給程式，例如 javac 本身就附帶了一堆引數可以使用，直接鍵入 javac 就可以顯示每個引數所代表的意義。



```
C:\>javac
Usage: javac <options> <source files>
where possible options include:
  -g                                Generate all debugging info
  -g:none                           Generate no debugging info
  -g:<lines,vars,source>           Generate only some debugging info
  -nowarn                           Generate no warnings
  -verbose                           Output messages about what the compiler is doing
  -deprecation                      Output source locations where deprecated APIs are u
sed
  -classpath <path>                 Specify where to find user class files
  -cp <path>                         Specify where to find user class files
  -sourcepath <path>                Specify where to find input source files
  -bootclasspath <path>             Override location of bootstrap class files
  -extdirs <dirs>                   Override location of installed extensions
  -endorseddirs <dirs>              Override location of endorsed standards path
  -d <directory>                   Specify where to place generated class files
  -encoding <encoding>              Specify character encoding used by source files
  -source <release>                 Provide source compatibility with specified release

  -target <release>                Generate class files for specific VM version
  -version                          Version information
  -help                            Print a synopsis of standard options
  -X                               Print a synopsis of nonstandard options
  -J<flag>                         Pass <flag> directly to the runtime system
```

圖 6.7 javac 可附帶的引數說明

在文字模式下啟動一個 Java 程式時，您也可以一併指定引數，以讓程式進行相對應的功能，也就是輸入命令列引數（Command line argument）給程式，在您撰寫主程式時，會在 main() 的參數列撰寫 String[] args，目的就是用來接受一個字串陣列，您只要取得 args 中的元素值，就可以取出 Java 程式運行時被指定的引數，範例 6.6 是個簡單的示範。

範例 6.6 CommandLineArg.java

```
public class CommandLineArg {
    public static void main(String[] args) {
        System.out.print("讀入的引數: ");
        for(int i = 0; i < args.length; i++) {
            System.out.print(args[i] + " ");
        }
        System.out.println();
    }
}
```

args 索引 0 的值是從程式名稱後第一個引數開始，以空白為區隔，依序地儲存在 args 陣列中，執行的方式與結果如下頁：

```
java CommandLineArg -file student.dat
讀入的引數: -file student.dat
```

當然您也可以使用 J2SE 5.0 新增的 foreach 語法來改寫範例 6.6，以循序取出被輸入的引數，範例 6.7 的執行結果與範例 6.6 是相同的。

範例 6.7 CommandLineArg2.java

```
public class CommandLineArg2 {
    public static void main(String[] args) {
        System.out.print("讀入的引數: ");
        for(String arg : args)
            System.out.print(arg + " ");
        System.out.println();
    }
}
```

良葛格的話匣子 在寫 main() 方法中的參數列時，也可以寫成 String args[], 這是陣列宣告取自於 C/C++ 的寫法，不過一般建議寫成 String[] args，比較符合 Java 的陣列宣告語法。由於命令列引數會傳遞給 args 陣列，如果您要存取 args 陣列的話，記得要檢查陣列長度，否則使用者若沒提供引數的話，會發生 ArrayIndexOutOfBoundsException 例外，一個檢查使用者是否有輸入引數的方式是檢查 args 陣列的長度，例如：

```
if(args.length == 0) {
    // 使用者沒有指定引數，顯示引數功能畫面
}
else {
    // 執行所指定引數的對應功能
}
```

在第10章學到例外處理之後，您還可以使用try...catch語法來取代if判斷式。

6.2.2 分離字串

將字串依所設定的條件予以分離是很常見的操作，例如指令的分離、文字檔案的資料讀出等，以後者而言，當您 在文字檔案中儲存以下的資料時，在讀入檔案後，將可以使用 String 的 split() 來協助每一格的資料分離。

```
justin    64/5/26    0939002302    5433343
momor    68/7/23    0939100391    5432343
```

範例 6.8 是個簡單的示範，假設 fakeFileData 的資料就是檔案中讀入的文字資料。

範例 6.8 SplitStringDemo.java

```
public class SplitStringDemo {
    public static void main(String args[]) {
        String[] fakeFileData = {
            "justin\t64/5/26\t0939002302\t5433343",
            "momor\t68/7/23\t0939100391\t5432343" };
        for(String data : fakeFileData) {
            String[] tokens = data.split("\t");
            for(String token : tokens) {
                System.out.print(token + "\t| ");
            }
            System.out.println();
        }
    }
}
```

```

        }
    }
}

```

執行結果：

```

justin | 64/5/26      | 0939002302   | 5433343     |
momor  | 68/7/23      | 0939100391   | 5432343     |

```

split() 依您所設定的分隔設定，將字串分為數個子字串並以 String 陣列傳回，這邊簡單的介紹了一下 split() 方法的使用，有些用過 Java 的人可能會想到 java.util.StringTokenizer，基本上 API 文件中明確的表示 StringTokenizer 已經是「遺產類別」（Legacy class）了，存在的原因是為了與舊版 Java 程式的相容性，不建議在您撰寫新的 Java 程式時使用，使用 split() 來代替會是個好的方案，而且您還可以進一步搭配正則表示式來進行字串分離。

6.2.3 使用正則表示式（Regular expression）

如果您查詢 J2SE 1.4 之後的 String 線上 API 手冊說明，您會發現有 matches()、replaceAll() 等方法，您所傳入的引數是「正則表示式」（Regular expression）的字串，正則表示式最早是由數學家 Stephen Kleene 于 1956 年提出，主要使用在字元字串的格式比對，後來在資訊領域廣為應用，現在已經成為 ISO（國際標準組織）的標準之一。

Java 在 J2SE 1.4 之後開始支援正則表示式，您可以在 API 文件的 java.util.regex.Pattern 類別中找到支援的正則表示式相關資訊；您可以將正則表示式應用於字串的比對、取代、分離等動作上，以下將介紹幾個簡單的正則表示式。

對於一些簡單的字元比對，例如 1 到 9、A-Z 等，您可以使用預先定義的符號來表示，表 6.4 列出幾個常用的字元比對符號。

表 6.4 字元比對符號

方法	說明
.	符合任一字元
\d	符合 0 到 9 任一個數字字元
\D	符合 0-9 以外的字元
\s	符合 '\t'、'\n'、'\x0B'、'\f'、'\r' 等空白字元
\w	符合 a 到 z、A 到 Z、0 到 9 等字元，也就是數字或是字母都符合
\W	符合 a 到 z、A 到 Z、0 到 9 等之外的字元，也就是除數字與字母外都符合

舉例來說，如果有一字串 "abcdebcadxbc"，使用 ".bc" 來作比對的話，符合的子字串有 "abc"、"ebc"、"xbc" 三個；如果使用 "..cd"，則符合的子字串只有 "abcd"，範例 6.9 證實這個說明。

範例 6.9 RegularExpressionDemo.java

```

public class RegularExpressionDemo {
    public static void main(String[] args) {
        String text = "abcdebcadxbc";

        String[] tokens = text.split(".bc");
        for(String token : tokens) {
            System.out.print(token + " ");
        }
        System.out.println();

        tokens = text.split("..cd");
        for(String token : tokens) {

```

```

        System.out.print(token + " ");
    }
    System.out.println();
}

}

```

執行結果：

```
d ad
ebcadxbc
```

使用 ".bc" 來作比對的話，由於符合的子字串有 "abc"、"ebc"、"xbc" 三個，所以 split() 方法會使用這三個字串為依據來作字符串分離，傳回的自然就是不符合表示式 ".bc" 的 "d" 與 "ad"，同理如果表示式為 "..cd"，則使用 split() 傳回的就是不符合 "..cd" 的 "ebcadxbc"。

您也可以使用「字元類」（Character class）來比較一組字元範圍，表 6.5 示範了幾個字元類的設定方式。

表 6.5 字元類範例

範例	作用
[abc]	符合 a、b 或 c
abc	符合「a 或 b 或 c」之外的字元
[a-zA-Z]	符合 a 到 z 或者是 A 到 Z 的字元
[a-d[m-p]]	a 到 d 或者是 m 到 p，也可以寫成 [a-dm-p]
[a-z&&[def]]	a 到 z 並且是 d 或 e 或 f，結果就是 d 或 e 或 f 可以符合
[a-z&& bc]	a 到 z 並且不是 b 或 c
[a-z&& m-p]	a 到 z 並且不是 m 到 p

指定一個字元之外，您也可以加上「貪婪量詞」（Greedy quantifiers）來指定字元可能出現的次數，表 6.6 示範了幾個例子。

表 6.6 貪婪量詞範例

範例	作用
X?	X 可出現一次或完全沒有
X*	X 可出現零次或多次
X+	X 可出現一次或多次
X{n}	X 可出現 n 次
X{n,}	X 可出現至少 n 次
X{n, m}	X 可出現至少 n 次，但不超過 m 次
X?	X 可出現一次或完全沒有

另外還有 Reluctant quantifiers、Possessive quantifiers 等的指定，您可以自行參考 `java.util.regex.Pattern` 類別 API 文件中的說明。

在 String 類別中，`matches()` 方法可以讓您驗證字串是否符合指定的正則表示式，這通常用於驗證使用者輸入的字串資料是

否正確，例如電話號碼格式；`replaceAll()`方法可以將符合正則表示式的子字串置換為指定的字串；`split()`方法可以讓您依指定的正則表示式，將符合的子字串排除，剩下的子字串分離出來並以字串陣列傳回，範例 6.9 已經示範了`split()`方法的使用，接下來在範例 6.10 中示範了`replaceAll()`與`matches()`方法的運用。

範例 6.10 UseRegularExpression.java

```
import java.io.*;

public class UseRegularExpression {
    public static void main(String args[])
        throws IOException {
        BufferedReader reader =
            new BufferedReader(
                new InputStreamReader(System.in));

        System.out.println("abcdefgabcabc".replaceAll(".bc", "###"));

        String phoneEL = "[0-9]{4}-[0-9]{6}";
        String urlEL = "<a.+href*=*['\"]?.*?[\'\"]?.*?>";
        String emailEL = "^[_a-zA-Z0-9-]+(.[_a-zA-Z0-9-]+)*@"
            + "[a-zA-Z0-9-]+([.][a-zA-Z0-9-]+)*$";

        System.out.print("輸入手機號碼：");
        String input = reader.readLine();

        if(input.matches(phoneEL))
            System.out.println("格式正確");
        else
            System.out.println("格式錯誤");

        System.out.print("輸入href標籤：");
        input = reader.readLine();

        // 驗證href標籤
        if(input.matches(urlEL))
            System.out.println("格式正確");
        else
            System.out.println("格式錯誤");

        System.out.print("輸入電子郵件：");
        input = reader.readLine();

        // 驗證電子郵件格式
        if(input.matches(emailEL))
            System.out.println("格式正確");
        else
            System.out.println("格式錯誤");
    }
}
```

執行結果：

```
#####defg#####
輸入手機號碼： 0939-100391
格式正確
輸入href標籤： <a href="http://caterpillar.onlyfun.net">
格式正確
輸入電子郵件： caterpillar.onlyfun@gmail.com
格式正確
```

良葛格的話匣子 正則表示式的設計是門學問，也有專書就是在介紹正則表示式的設計，沒有經常使用的話，很難設計出實用性高的正則表示式，這邊只能說大致介紹而已，如果真有需要某種正則表示式，建議可以使用搜尋引擎找看看有無現成或類似的表示式可以使用，要不然的話，就只有找專書研究研究如何設計了。

6.2.4 Pattern、Matcher

String 上可使用正則表示式的操作，實際上是利用了 `java.util.regex.Pattern` 與 `java.util.regex.Matcher` 的功能，當您呼叫 `String` 的 `matches()` 方法時，實際上是呼叫 `Pattern` 的靜態方法 `matches()`，這個方法會傳回 `boolean` 值，表示字串是否符合正則表示式。

如果您想要將正則表示式視為一個物件來重複使用，則您可以使用 `Pattern` 的靜態方法 `compile()` 進行編譯，`compile()` 方法會傳回一個 `Pattern` 的實例，這個實例代表您的正則表示式，之後您就可以重複使用 `Pattern` 實例的 `matcher()` 方法來傳回一個 `Matcher` 的實例，代表符合正則式的實例，這個實例上有一些尋找符合正則式條件的方法可供操作，範例 6.11 直接作了示範。

範例 6.11 UsePatternMatcher.java

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

public class UsePatternMatcher {
    public static void main(String[] args) {
        String phones1 =
            "Justin 的手機號碼 : 0939-100391\n" +
            "momor 的手機號碼 : 0939-666888\n";

        Pattern pattern = Pattern.compile(".*0939-\d{6}");
        Matcher matcher = pattern.matcher(phones1);

        while(matcher.find()) {
            System.out.println(matcher.group());
        }

        String phones2 =
            "caterpillar 的手機號碼 : 0952-600391\n" +
            "bush 的手機號碼 : 0939-550391";

        matcher = pattern.matcher(phones2);

        while(matcher.find()) {
            System.out.println(matcher.group());
        }
    }
}
```

範例 6.11 會尋找手機號碼為 0939 開頭的號碼，假設您的號碼來源不只一個（如 `phones1`、`phones2`），則您可以編譯好正則表示式並傳回一個 `Pattern` 物件，之後就可以重複使用這個 `Pattern` 物件，在比對時您使用 `matcher()` 傳回符合條件的 `Matcher` 實例，`find()` 方法表示是否有符合的字串，`group()` 方法則可以將符合的字串傳回，程式的執行結果如下：

```
Justin 的手機號碼 : 0939-100391
momor 的手機號碼 : 0939-666888
bush 的手機號碼 : 0939-550391
```

來使用 `Pattern` 與 `Matcher` 改寫一下範例 6.9，讓程式可以傳回符合正則式的字串，而不是傳回不符合的字串。

範例 6.12 RegularExpressionDemo2.java

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

public class RegularExpressionDemo2 {
    public static void main(String[] args) {
        String text = "abcdebcadxbc";

        Pattern pattern = Pattern.compile(".bc");
        Matcher matcher = pattern.matcher(text);

        while(matcher.find()) {
            System.out.println(matcher.group());
        }
        System.out.println();
    }
}
```

```
    }  
}
```

執行結果：

```
abc  
ebc  
xbc
```

良葛格的話匣子 在這邊想再嘮叨一下，在書中我有介紹的類別上之方法操作，都只是 API 中一小部份而已，目的只是讓您瞭解有這個功能的存在，並以實際的範例體會其功能，真正要瞭解每個方法的操作，「請多參考線上 API 文件」，我不想列出一長串的表格來說明每個方法，這只會讓篇幅增加，也只會模糊了您學習的焦點，而且請記得，學會查詢 API 文件，絕對是學好 Java 的必備功夫。

6.3 接下來的主題

每一個章節的內容由淺至深，初學者該掌握的深度要到哪呢？在這個章節中，對於初學者我建議至少掌握以下幾點內容：

- 會使用 String 類別上的常用方法
- 瞭解字串的不可變數變動（immutable）特性
- 知道有字串池的存在
- 瞭解使用 '==' 與 equals() 方法比較兩個字串時的不同點
- 瞭解 '+' 使用於字串串接上的負擔

到目前為止，您僅止於使用物件的階段，接下來就要進入重頭戲了，從下一個章節開始，您將會逐步接觸 Java 中物件導向程式設計的領域，瞭解 Java 中的何種特性可以支援物件導向設計。

第 7 章 封裝 (Encapsulation)

從這個章節開始，您將逐步學習如何使用 Java 在「物件導向」（Object-oriented）上的語法支援，以進行物件導向程式設計，在此之前請先記得一個觀念：「學會一個支援物件導向的程式語言（如 Java）與學會物件導向（程式設計）觀念完全是兩碼子事。」物件導向是一種對問題的思考方式，與任何的程式語言沒有任何直接的關係，物件導向也絕不僅用於程式設計領域。

物件導向可以作為一門學科來討論，要理解並應用它並不容易，更不是在幾個章節內就可以詳細說明的，如果您沒有接觸過物件導向，光是討論觀念會過於抽象，「從實作中學習」會是比較好的方式，所以我會在說明 Java 對物件導向上的支援語法時，同時說明一些物件導向的入門觀念，首先第一步是瞭解如何使用 Java 來描述物件的特性，對物件資訊進行「封裝」（Encapsulation），也就是定義「類別」（Class）。

7.1 定義類別 (Class)

以物件導向的思維來思考一個問題的解答時，會將與問題相關的種種元素視作一個個的物件，問題的發生是由物件的交互所產生，而問題的解答也由某些物件彼此合作來完成。所以如何描述問題中的各種元素？如何將這些元素定義為物件？也就是如何封裝物件資訊就是物件導向設計的第一步。您要瞭解如何使用「類別」（Class）定義物件，類別是建構物件時所依賴的規格書。

7.1.1 以物件思考問題

簡單的說，物件導向的思維就是以物件為中心來思考問題，然而什麼又叫作「以物件為中心來思考問題」？我不想用太多抽象的字眼來解釋這些詞語，這邊實際提出一個問題，並嘗試以物件導向的方式來思考問題。

有一個帳戶，帳戶中有存款餘額，您可以對帳戶進行存款與提款的動作，並可以查詢以取得存款餘額。

要以物件為中心來思考問題，首先要識別出問題中的物件，以及物件上的屬性與可操作的方法：

- 識別問題中的物件與屬性

帳戶是個比較單純的問題，可以從問題中出現的名詞來識別出物件，描述中有「帳戶」與「餘額」兩個名詞，基本上兩個名詞都可以識別成物件，然而在這個簡單的問題當中，設計的粒度還不需要這麼細，所以您先識別「帳戶」這個物件。

識別出物件之後，接下來看看物件上有什麼屬性（Property），像是物件上擁有什么特徵或是可表示的狀態（State），屬性是物件上的靜態特性。屬性基本上也可以從名詞上識別，在這個例子中，您可以將「餘額」作為帳戶的屬性之一。

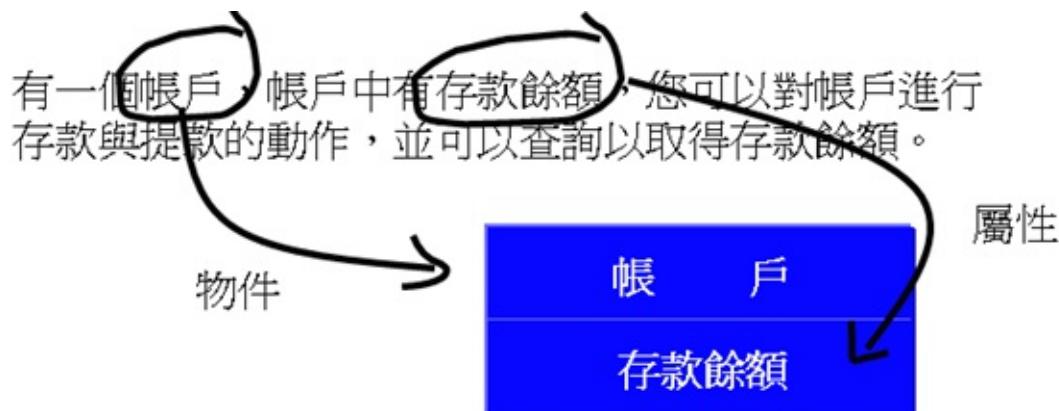


圖 7.1 識別出物件與屬性

- 識別物件上的方法

接著要識別物件上的方法，也就是識別物件上的動態特性，也就是物件本身可操作或供操作的介面，問題描述上的動詞可能就可以識別為方法，例如「存款」、「提款」、「查詢餘額」等動作，就可以識別為物件上的方法。

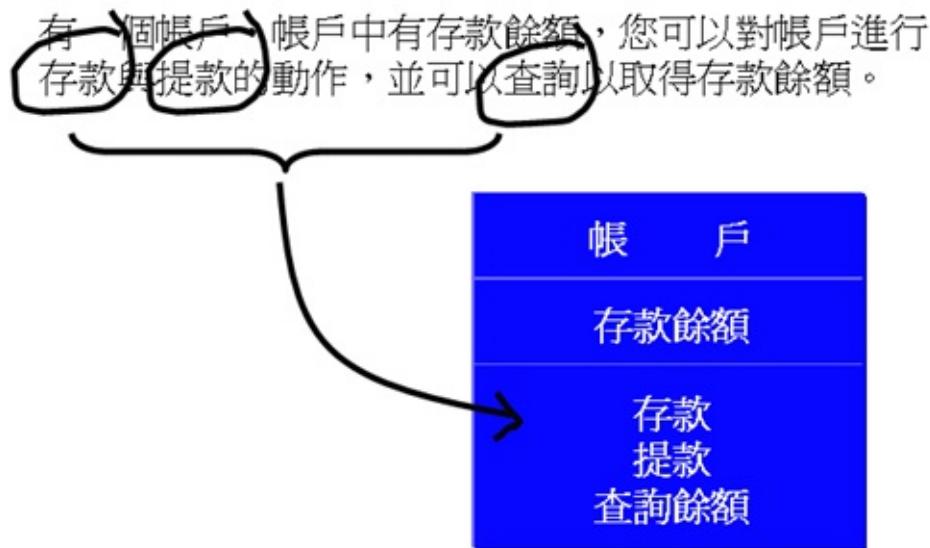


圖 7.2 識別物件方法

識別出物件及其上的屬性與方法之後，您就有了基本的物件定義書，接著您就可以實際從定義書中產生物件實例，並以這些物件實例設計彼此間的交互行為以解決問題。

有一個帳戶，帳戶中有存款餘額，您可以對帳戶進行
存款與提款的動作，並可以查詢以取得存款餘額。

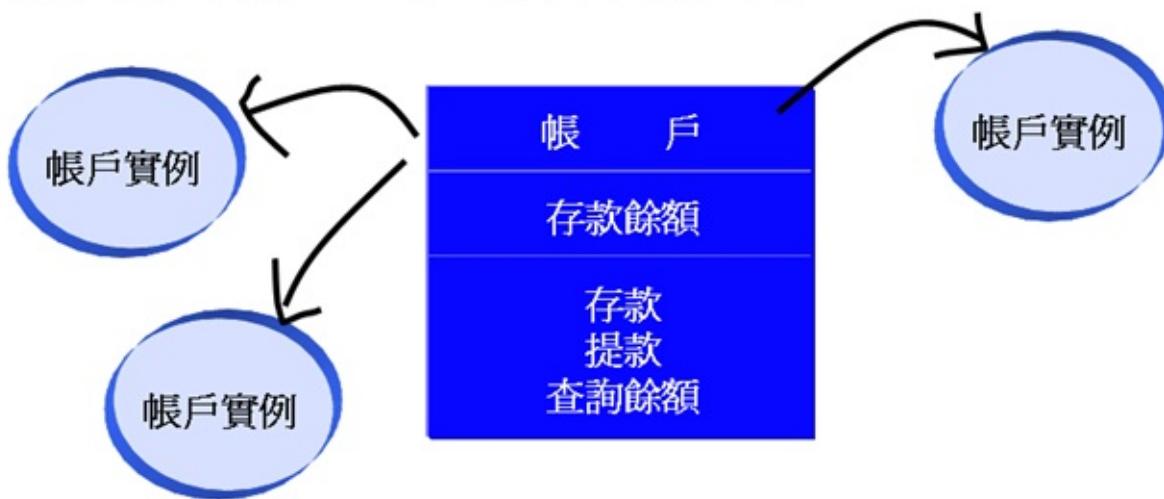


圖 7.3 從物件定義書中產生物件實例

如何為這些物件實例設計交互行為，依應用的領域不同而有所差異，就 Java 程式設計而言，就是使用 Java 的語法來為這些物件來進行各種條件判斷與流程控制，接著運行程式以獲得解答。

良葛格的話匣子 以上是很簡單的物件分析過程，目的在讓您對物件導向分析有大致的瞭解，對於真正所面臨的問題，實際的物件分析會再複雜一些，例如單純從問題中的名詞來識別物件就不一定行的通了，這與「物件導向分析」(Object-oriented Analysis) 有關，如果您想進一步瞭解物件導向分析，建議看看這本書：

Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process,
Second Edition By Craig Larman

7.1.2 使用 class 定義類別

在物件導向設計中，物件並不會憑空產生，您必須識別出問題中的物件，並對這些物件加以定義，您要定義一個規格書，在定義類別 (Class)

Java 中這個規格書稱之為「類別」（Class），您使用類別定義出物件的規格書，之後根據類別來建構出一個個的物件，然後透過物件所提供的操作介面來與程式互動。

在 Java 中使用 "class" 關鍵字來定義類別，使用類別來定義一個物件（Object）時，會考慮這個物件可能擁有的「屬性」（Property）與「方法」（Method）。屬性是物件的靜態表現，而方法則是物件與外界互動的動態操作。

舉個例子來說，若您的問題中會有「帳戶」這個物件，在分析了您的問題之後，您為「帳戶」這個物件定義了 Account 類別。

範例 7.1 Account.java

```
public class Account {
    private String accountNumber;
    private double balance;

    public Account() {
        this("empty", 0.0);
    }

    public Account(String accountNumber, double balance) {
        this.accountNumber = accountNumber;
        this.balance = balance;
    }

    public String getAccountNumber() {
        return accountNumber;
    }

    public double getBalance() {
        return balance;
    }

    public void deposit(double money) {
        balance += money;
    }

    public double withdraw(double money) {
        balance -= money;
        return money;
    }
}
```

在這邊先大致呈現出一個類別所可能具備的元素並加以簡介，稍後對每個元素會詳加介紹：

- 定義類別

首先看到範例中的 "class"，這是 Java 中用來定義類別的關鍵字，記得一個類別的定義是這麼作的：

```
public class Account {
    // 實作內容
}
```

Account 是您為類別取的名稱，由於這個類別使用 "public" 關鍵字加以修飾，所以檔案的主檔名必須與類別名稱相同，也就是檔案要取名為 "Account.java"，這是規定，在一個檔案中可以定義數個類別，但只能有一個類別被設定為 "public"，檔案名稱主檔名必須與這個 public 的類別同名，例如 Account.java 中可以有以下的內容：

```
public class Account { // 檔案必須是Account.java
    // 實作內容
}
class SomeClass {
    // 實作內容
}
class OtherClass {
```

```
// 實作內容
}
```

- 定義成員

在類別中的資料及互動方法，統稱其為「類別成員」（Class member），範例 7.1 中的 accountNumber、balance 成員是「資料成員」（Field member），getAccountNumber() 與 getBalance() 是「方法成員」（Method member），在定義資料成員時可以指定初值，如果沒有指定初值，則會有預設值，資料成員如果是基本型態，則預設值與表 5.1 所列出的相同，如果是物件型態，則預設值為 null，也就是不參考任何的物件。

注意到 "public" 這個關鍵字，這表示所定義的成員可以使用宣告的物件名稱加上 '.' 運算子來直接呼叫，也稱之為「公用成員」或「公開成員」。"private" 這個關鍵字用來定義一個「私用成員」，私用成員不可以透過參考名稱加上 "." 直接呼叫，又稱之為「私有成員」。

在定義類別時，有一個基本原則是：資訊的最小化公開。也就是說儘量透過方法來操作物件，而不直接存取物件內部的資料成員（也就是 Field 成員）。資訊的最小化公開原則是基於安全性的考量，避免程式設計人員隨意操作內部資料成員而造成程式的錯誤，您可以在日後的程式設計中慢慢來體會；在稍後的實作中，您將看到我不會對 accountNumber 與 balance 兩個私用成員直接存取，而會透過公開的方法來對它們進行設定。

一個類別中的資料成員，若宣告為 "private"，則其可視範圍（Scope）為整個類別內部，由於外界無法直接存取私用成員，所以您要使用兩個公開方法 getAccountNumber() 與 getBalance() 分別傳回其這兩個成員的值。

- 定義建構方法

與類別名稱同名的方法稱之為「建構方法」（Constructor），也有人稱之為「建構子」，它沒有傳回值，建構方法的作用是讓您建構物件的同時，可以同時初始一些必要的資訊，建構方法可以被「重載」（Overload），以滿足物件生成時各種不同的初始需求，在範例 7.1 中您重載了建構方法，在不指定引數的情況下，會將 balance 設定為 0.0，而 accountNumber 設定為 "empty"，另一個建構方法則可以指定引數，this() 方法用於物件內部，表示呼叫物件的建構方法，另一個關鍵字就是 "this"，它參考至物件本身，7.1.5 會再詳細介紹 "this" 以進一步瞭解其作用。

定義好 Account 類別之後，您就可根據這個類別來建構物件，也就是產生 Account 類別的實例，建構物件時要使用 "new" 關鍵字，顧名思義，就是根據所指定的類別（規格書）「新建」一個物件：

```
Account account1 = new Account();
Account account2 = new Account("123-4567", 100.0);
```

在上面的程式片段中宣告了 account1 與 account2 兩個 Account 型態的參考名稱，並讓它們分別參考至物件，account1 所參考的物件在建立時並不指定任何引數，所以根據之前對 Account 類別的定義，account1 所參考物件的 balance 將設定為 0.0，accountNumber 設定為 "empty"；account2 所參考的物件在新建時則給定兩個引數，所以 account2 所參考物件的 balance 設定為 100.0，而 accountNumber 設定為 "empty"。

要透過公開成員來操作物件或取得物件資訊的話，可以在物件名稱後加上「.」運算子來進行，例如：

```
account1.getBalance();
account1.deposit(1000.0);
```

範例 7.2 綜合以上的介紹來作個簡單的練習，要使用到範例 7.1 中的 Account 類別。

範例 7.2 AccountDemo.java

```
public class AccountDemo {
    public static void main(String[] args) {
        Account account = new Account();
```

```

        System.out.println("帳戶: " + account.getAccountNumber());
        System.out.println("餘額: " + account.getBalance());

        account = new Account("123-4567", 100.0);
        account.deposit(1000.0);
        System.out.println("帳戶: " + account.getAccountNumber());
        System.out.println("餘額: " + account.getBalance());
    }
}

```

Account.java 與 AccountDemo.java 都要編譯，然後執行程式，結果如下：

```

帳戶: empty
餘額: 0.0
帳戶: 123-4567
餘額: 1100.0

```

良葛格的話匣子 類別與物件這兩個名詞會經常混於書籍與文件之中，例如「您可以使用 Scanner 類別」、「您可以使用 Scanner 物件」，這兩句在某些場合其意思可能是相同的，不過要細究的話，兩句的意思通常都是「您可以使用根據 Scanner 類別所建構出來的物件」，不過寫這麼長很煩，難免就省略了一些字眼。

7.1.3 類別成員 (Class member)

在 Java 中，一個類別可以定義資料成員 (Field) 及方法 (Method) 成員，在 Java 中，類別成員可用的存取權限修飾詞有 "public"、"protected"、"private" 三個，如果在宣告成員時不使用存取修飾詞，則預設以「套件」 (package) 為存取範圍，也就是說在 package 外就無法存取，關於 package 與存取修飾的關係，在第 9 章還會見到說明。

以範例 7.1 為例來進行說明。在該範例中，您定義了一個 Account 類別，當中還定義了 accountNumber 與 balance 兩個資料成員，這兩個資料成員被宣告為 "private"，表示它是「私用成員」 (Private member)，私用成員只能在 Account 類別中被使用，不可以直接藉由物件的參考名稱加上 "." 來直接存取它。

再來看到方法 (Method) 成員，範例 7.1 的每一個方法被宣告為 "public"，表示這些方法可以藉由物件的參考名稱加上 "." 直接呼叫，一個方法成員為一小個程式片段或一個執行單元 (Unit)，這個程式片段可重複被呼叫使用，並可傳入引數或傳回一個表示執行結果的數值，一個方法成員的基本宣告與定義方式如下：

```

存取修飾 傳回值型態 方法名稱(參數列) {
    // 實作
    return 傳回值;
}

```

參數列用來傳入方法成員執行時所需的資料，如果傳入的引數是基本資料型態 (Primitive data type)，則會將值複製至參數列上的參數，如果傳入的引數是一個物件，則會將參數列上宣告的參數參考至指定的物件。

方法區塊中可以宣告變數 (Variable)，參數在方法區塊執行結束後就會自動清除，如果方法中宣告的變數名稱與類別資料成員的名稱同名，則方法中的變數名稱會暫時覆蓋資料成員的作用範圍；參數列上的參數名稱也會覆蓋資料成員的作用範圍，如果此時要在方法區塊中使用資料成員，可以使用 "this" 關鍵字來特別指定，範例 7.3 可以印證這個說明。

範例 7.3 MethodMember.java

```

public class MethodMember {
    public static void main(String[] args) {
        MethodDemo methodDemo = new MethodDemo();

        methodDemo.scopeDemo(); // 對 data 資料成員不會有影響
        System.out.println(methodDemo.getData());

        methodDemo.setData(100); // 對 data 資料成員不會有影響
    }
}

```

```

        System.out.println(methodDemo.getData());
    }

}

class MethodDemo {
    private int data = 10;

    public void scopeDemo() { // void 表示沒有傳回值
        int data = 100;
    }

    public int getData() {
        return data;
    }

    public void setData(int data) { // void 表示沒有傳回值
        data = data; // 這樣寫是沒用的
        // 寫下面這個才有用
        // this.data = data;
    }
}

```

執行結果：

```

10
10

```

方法的傳回值可以將計算的結果或其它想要的數值、物件傳回，傳回值與傳回值型態的宣告必須一致，在方法中如果執行到 "return" 陳述，則會立即終止區塊的執行；如果方法執行結束後不需要傳回值，則可以撰寫 "void"，且無需使用 "return" 關鍵字。

在物件導向程式設計的過程中，有一個基本的原則，如果資料成員能不公開就不公開，在 Java 中若不想公開成員的資訊，方式就是宣告成員為 "private"，這是「資訊的最小化」，此時在程式中要存取 "private" 成員，就要經由 setXXX() 與 getXXX() 等公開方法來進行設定或存取，而不是直接存取資料成員。

透過公開方法存取私用成員的好處之一是，如果存取私用成員的流程有所更動，只要在公開方法中修改就可以了，對於呼叫方法的應用程式不受影響，例如您的 Account 類別中，withdraw() 顯然的在餘額為 0 時，仍然可以提款，您必須對此做出修正：

```

public double withdraw(double money) {
    if(balance - money < 0) {
        return 0;
    }
    else {
        balance -= money;
        return money;
    }
}

```

這麼一來，您的 withdraw() 對 balance 做了些檢查，但對於使用 Account 的 AccountDemo 來說，並不用做出修改。

在第 4 章中介紹過 autoboxing、unboxing，在方法的參數列中是可以作用的，也就是說如果您的方法中是這樣設計的：

```

public class SomeClass {
    ...
    public void someMethod(Integer integer) {
        ....
    }
    ...
}

```

則您可以使用這樣的方式來設定引數：

```
SomeClass someObj = new SomeClass();
someObj.someMethod(1); // autoboxing
```

良葛格的話匣子 方法名稱的命名慣例為首字小寫，名稱以一目瞭解方法的作用為原則，以上所採取的都是駱駝式的命名方式，也就是每個單字的首字予以適當的大寫，例如 someMethodOfSomeClass() 這樣的命名方式。

為資料成員設定 setXXX() 或 getXXX() 存取方法時，XXX 名稱最好與資料成員名稱相對應，例如命名balance這個資料成員對應的方法時，可以命名為 setBalance() 與 getBalance()，而 accountNumber 這個成員，則可對應於 setAccountNumber() 與 getAccountNumber() 這樣的名稱，如此閱讀程式時可以一目瞭解方法的存取對象。

您搞得清楚「參數」（Parameter）與引數（Argument）嗎？在定義方法時，可以定義「參數列」，例如：

```
public void setSomething(int something) { // something 稱之為參數
    // ...
}
```

而呼叫方法時傳遞的數值或物件稱之為「引數」，例如：

```
someObject.setSomething(10); // 10 是引數
```

7.1.4 建構方法（Constructor）

在定義類別時，您可以使用「建構方法」（Constructor）來進行物件的初始化，在 Java 中建構方法是與類別名稱相同的公開方法成員，且沒有傳回值，例如：

```
public class SafeArray {
    // ...
    public SafeArray() { // 建構方法
        // ...
    }
    public SafeArray(參數列) { // ...
        // ...
    }
}
```

在建構方法中，您可以定義無參數的或具有參數的建構方法，程式在運行時，會根據配置物件時所指定的引數資料型態等，來決定該使用哪一個建構方法新建物件，如果您沒有定義任何的建構方法，則編譯器會自動配置一個無參數且沒有陳述內容的建構方法。

在範例 7.4 示範了實作簡單的「安全的陣列」，您所定義的陣列類別可以動態配置陣列長度，並可事先檢查存取陣列的索引是否超出陣列長度，在這個陣列類別中，您還實作了幾個簡單的功能，像是傳回陣列長度、設定陣列元素值、取得陣列元素值等。

範例 7.4 SafeArray.java

```
public class SafeArray {
    private int[] arr;

    public SafeArray() {
        this(10); // 預設 10 個元素
    }
}
```

```

public SafeArray(int length) {
    arr = new int[length];
}

public void showElement() {
    for(int i : arr) {
        System.out.print(i + " ");
    }
}

public int getElement(int i) {
    if(i >= arr.length || i < 0) {
        System.err.println("索引錯誤");
        return 0;
    }

    return arr[i];
}

public int getLength() {
    return arr.length;
}

public void setElement(int i, int data) {
    if(i >= arr.length || i < 0) {
        System.err.println("索引錯誤");
        return;
    }

    arr[i] = data;
}
}

```

如果您不指定引數的話，就會使用無參數的建構方法來配置 10 個元素的陣列，您也可以由指定的長度來配置陣列；您在無參數的建構方法中使用 `this(10)`，這會呼叫另一個有參數的建構方法，以避免撰寫一些重複的原始碼。範例 7.5 示範了如何使用自訂的安全陣列類別。

範例 7.5 SafeArrayDemo.java

```

public class SafeArrayDemo {
    public static void main(String[] args) {
        // 預設10個元素
        SafeArray arr1 = new SafeArray();
        // 指定配置 5 個元素
        SafeArray arr2 = new SafeArray(5);

        for(int i = 0; i < arr1.getLength(); i++)
            arr1.setElement(i, (i+1)*10);

        for(int i = 0; i < arr2.getLength(); i++)
            arr2.setElement(i, (i+1)*10);

        System.out.print("arr1: ");
        arr1.showElement();

        System.out.print("\narr2: ");
        arr2.showElement();
    }
}

```

在範例 7.5 中您配置了兩個物件，一個使用預設的建構方法，所以 `arr1` 的陣列元素會有 10 個，一個使用指定長度的建構方法，所以 `arr2` 的陣列元素個數是您指定的5，建構方法依引數不同而自行決定該使用哪一個建構方法，執行結果如下：
`arr1: 10 20 30 40 50 60 70 80 90 100`
`arr2: 10 20 30 40 50`

7.1.5 關於 `this`

請您回顧一下範例 7.1，在範例的 `Account` 類別中定義有 `accountNumber` 與 `balance` 成員，當您使用 `Account` 類別新增兩

個物件並使用 account1 與 account2 來參考時，account1 與 account2 所參考的物件會各自擁有自己的 accountNumber 與 balance 資料成員，然而方法成員在記憶體中會只有一份，當您使用 account1.getBalance() 與 account1.getBalance() 方法取回 balance 的值時，既然類別的方法成員只有一份，getBalance() 時如何知道它傳回的 balance 是 account1 所參考物件的 balance，還是 account2 所參考物件的 balance 呢？

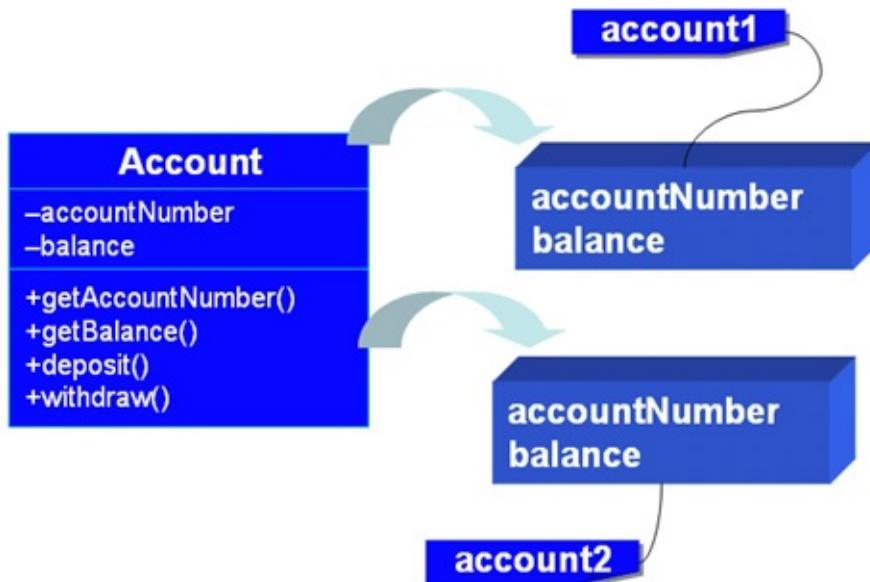


圖 7.4 物件實例擁有自己的資料成員

其實您使用參考名稱來呼叫物件的方法成員時，程式會將物件的參考告知方法成員，而在方法中所撰寫的每一個資料成員其實會隱含一個 this 參考名稱，這個 this 名稱參考至呼叫方法的物件，當您呼叫 getBalance() 方法時，其實您相當於執行：

```
public double getBalance() {
    return this.balance;
}
```

所以當使用 account1 並呼叫 getBalance() 方法時，this 所參考的就是 account1 所參考的物件，而 account2 並呼叫 getBalance() 方法時，this 所參考的就是 account2 所參考的物件，所以 getBalance() 可以正確的得知該傳回哪一個物件的 balance 資料。

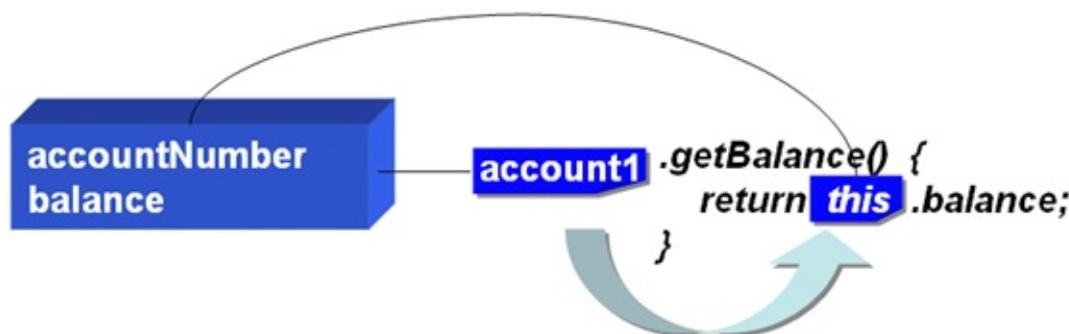


圖 7.5 this 參考至實際的物件

每一個類別的方法成員都會隱含一個 this 參考名稱，可用來指向呼叫它的物件，當您在方法中使用資料成員時，都會隱含的使用 this 名稱，當然您也可以明確的指定，例如在方法定義時使用：

```
public Account(String accountNumber, double balance) {
    this.accountNumber = accountNumber;
```

```

    this.balance = balance;
}

```

參數名稱與資料成員名稱相同時，為了避免參數的作用範圍覆蓋了資料成員的作用範圍，您必須明確的使用 this 名稱來指定，但如果參數名稱與資料成員名稱不相同則不用特別指定，例如：

```

public Account(String number, double money) {
    accountNumber = number; // 實際等於this.accountNumber = number;
    this.balance = money; // 實際等於this.balance = money;
}

```

this 除了用來參考至呼叫方法的實際物件之外，還有一種可以帶引數的用法，主要是用於呼叫建構方法，而避免直接以建構方法的名稱來呼叫，例如在下面的程式片段中，當使用無參數的建構方法 Ball() 時，它會呼叫有參數的建構方法：

```

public class Ball {
    private String name;

    public Ball() {
        this("No name"); // 會使用Ball("No name")來建構
    }

    public Ball(String name) {
        this.name = name;
        ...
    }
}

```

良葛格的話匣子 很多時候會經常這麼說：「account1 物件可以作 xxx 動作...」或是「account2 物件的 xxx 方法...」，其實意思指的是：「account1 所參考物件可以作 xxx 動作...」、「account2 所參考物件的 xxx 方法」，只不過每次都要這麼寫的話，會讓文件內容又臭又長，所以就都簡略的用 account1、account2 來代表物件了，但您自己要記得像 account1、account2 這樣的名稱，其目的是參考至實際的物件，這邊是因為要說明 this 的作用，所以要寫的詳細一些。

7.1.6 關於 static

對於每一個基於相同類別所產生的物件而言，它們會擁有各自的資料成員，然而在某些時候，您會想要這些物件擁有共享的資料成員，舉個例子來說，如果您設計了一個 Ball 類別，當中打算使用到圓周率PI這個資料，因為對於任一個 Ball 的實例而言，圓周率都是相同的，您不需要讓不同的 Ball 實例擁有各自的圓周率資料成員。

您可以將 PI 資料成員宣告為 "static"，被宣告為 "static" 的資料成員，又稱「靜態資料成員」，靜態成員是屬於類別所擁有，而不是個別的物件，您可以將靜態成員視為每個物件實例所共享的資料成員。要宣告靜態資料成員，只要在宣告資料成員時加上 "static" 關鍵字就可以了，例如：

```

public class Ball {
    public static double PI = 3.14159; // 宣告static資料
    ...
}

```

靜態成員屬於類別所擁有，可以在不使用名稱參考下，直接使用類別名稱加上'.'運算子來存取靜態資料成員，不過靜態資料成員同樣遵守 "public"、"protected" 與 "private" 的存取限制，所以若您要直接存取靜態資料成員，必須注意它的權限，例如必須設定為 "public" 成員的話就可以如下存取：

```

System.out.println("PI = " + Ball.PI);

```

雖然您也可以在宣告物件之後，透過物件名稱加上'.'運算子來存取靜態資料成員，但是這個方式並不被鼓勵，通常建議使用類別名稱加上'.'運算子來存取，一方面也可以避免與非靜態資料成員混淆，例如下面的方式是不被鼓勵的：

```
Ball ball = new Ball();
System.out.println("PI = " + ball.PI);
```

與靜態資料成員類似的，您也可以宣告方法成員為 "static" 方法，又稱「靜態方法」，被宣告為靜態的方法通常是作為工具方法，例如在 Ball 類別上增加一個角度轉徑度的方法 toRadian()：

```
public class Ball {
    ...
    public static double toRadian(double angle) {
        return 3.14159 / 180 * angle;
    }
}
```

與靜態資料成員一樣的，您可以透過類別名稱使用'.'運算子來存取 "static" 方法，當然要注意權限設定，例如設定為 "public" 的話可以如下存取：

```
System.out.println("角度90等於徑度" + Ball.toRadian (90));
```

靜態資料與靜態方法的作用通常是為了提供共享的資料或工具方法，例如將數學常用常數或計算公式，以 "static" 壓告並撰寫，之後您可以把這個類別當作工具類別，透過類別名稱來管理與取用這些靜態資料或方法，例如像 Java SE 所提供的 Math 類別上，就有 Math.PI 這個靜態常數，以及 Math.Exp()、Math.Log()、Math.Sin() 等靜態方法可以直接使用，另外還有像 Integer.parseInt()、Integer.MAX_VALUE 等也都是靜態方法與靜態資料成員的實際例子。

由於靜態成員是屬於類別而不是物件，所以當您呼叫靜態方法時，並不會傳入物件的參考，所以靜態方法中不會有 this 參考名稱，由於沒有 this 名稱，所以在 Java 的靜態方法中不允許使用非靜態成員，因為沒有 this 來參考至物件，也就無法辨別要存取的是哪一個物件的成員，事實上，如果您在靜態方法中使用非靜態資料成員，在編譯時就會出現以下的錯誤訊息：

```
non-static variable test cannot be referenced from a static context
```

或者是在靜態方法中呼叫非靜態方法，在編譯時就會出現以下的錯誤訊息：

```
non-static method showHello() cannot be referenced from a static context
```

在 Java 中程式進入點（Entry point）的 main() 方法就是靜態方法，如果您要直接在main()中呼叫其它的方法，則該方法就必須是靜態方法，像範例 7.6 所示範的。

範例 7.6 StaticDemo.java

```
public class StaticDemo {
    public static void sayHello() {
        System.out.println("哈囉！");
    }

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

您可以試著將 sayHello() 前的 "static" 消掉，編譯時就會發生上述的第二個錯誤訊息。

Java 在使用到類別時才會載入類別至程式中，如果在載入類別時，您希望先進行一些類別的初始化動作，您可以使用 "static" 定義一個靜態區塊，並在當中撰寫類別載入時的初始化動作，例如：

```
public class Ball {
    static {
        // 一些初始化程式碼
    }
    ...
}
```

在類別被載入時，預設會先執行靜態區塊中的程式碼，且只會執行一次，實際使用範例來說明一下，首先撰寫範例 7.7 的 SomeClass 類別。

範例 7.7 SomeClass.java

```
public class SomeClass {
    static {
        System.out.println("類別被載入");
    }
}
```

這個類別只定義了靜態區塊，主要是為了測試類別被載入時是否執行該區塊，接著撰寫測試程式，如範例 7.8 所示。

範例 7.8 StaticBlockDemo.java

```
public class StaticBlockDemo {
    public static void main(String[] args) {
        SomeClass c = new SomeClass();
    }
}
```

在使用 "new" 來建立 SomeClass 的實例時，SomeClass 類別會被載入，載入之後預設會執行靜態區塊的內容，所以程式的執行結果如下所示：

類別被載入

7.2 關於方法

在對定義類別有了瞭解之後，接下來再深入討論類別中的方法成員，在 Java 中，您可以「重載」（Overload）同名方法，而在 J2SE 5.0 之後，您還可以提供方法不定長度引數（Variable-length Argument），當然，最基本的您要知道遞迴（Recursive）方法的使用，最後還要討論一下 `finalize()` 方法，並從中瞭解一些 Java 「垃圾收集」（Garbage collection）的機制。

7.2.1 重載（Overload）方法

Java 支援方法「重載」（Overload），又有人譯作「超載」、「過載」，這種機制為類似功能的方法提供了統一的名稱，但可根據參數列的不同而自動呼叫對應的方法。

一個例子可以從 `String` 類別上提供的一些方法看到，像是 `String` 的 `valueOf()` 方法就提供了多個版本：

```
static String valueOf(boolean b)
static String valueOf(char c)
static String valueOf(char[] data)
static String valueOf(char[] data, int offset, int count)
static String valueOf(double d)
static String valueOf(float f)
static String valueOf(int i)
static String valueOf(long l)
static String valueOf(Object obj)
```

雖然您呼叫的方法名稱都是 `valueOf()`，但是根據所傳遞的引數資料型態不同，您會呼叫對應版本的方法來進行對應的動作，例如若是 `String.valueOf(10)`，因為 10 是 `int` 型態，所以會執行的方法是 `valueOf(int i)` 的版本，而若是 `String.valueOf(10.12)`，因為 10.12 是 `double` 型態，則會執行的方法是 `valueOf(double d)` 的版本。

方法重載的功能使得程式設計人員能較少苦惱於方法名稱的設計，以統一的名稱來呼叫相同功能的方法，方法重載不僅可根據傳遞引數的資料型態不同來呼叫對應的方法，參數列的參數個數也可以用來設計方法重載，例如您可以這麼重載 `someMethod()` 方法：

```
public class SomeClass {
    // 以下重載了someMethod()方法
    public void someMethod() {
        // ...
    }
    public void someMethod(int i) {
        // ...
    }
    public void someMethod(float f) {
        // ...
    }
    public void someMethod(int i, float f) {
        // ...
    }
}
```

要注意的是返回值型態不可用作為方法重載的區別根據，例如以下的方法重載是不正確的，編譯器仍會將兩個 `someMethod()` 視為重複的定義：

```
public class SomeClass {
    public int someMethod(int i) {
        // ...
        return 0;
    }
    public double someMethod(int i) {
        // ...
    }
}
```

```

        return 0.0;
    }
}

```

在 J2SE 5.0 後當您使用方法重載時，要注意到 autoboxing、unboxing 的問題，來看看範例 7.9，您認為結果會是什麼？

範例 7.9 OverloadTest.java

```

public class OverloadTest {
    public static void main(String[] args) {
        someMethod(1);
    }

    public static void someMethod(int i) {
        System.out.println("int 版本被呼叫");
    }

    public static void someMethod(Integer integer) {
        System.out.println("Integer 版本被呼叫");
    }
}

```

結果會顯示 "int 版本被呼叫"，您不能期待裝箱（boxing）的動作會發生，如果您想要呼叫參數列為 Integer 版本的方法，您要明確指定，例如：

```
someMethod(new Integer(1));
```

編譯器在處理重載方法、裝箱問題及「不定長度引數」時，會依下面的順序來尋找符合的方法：

- 找尋在還沒有裝箱動作前可以符合引數個數與型態的方法
- 嘗試裝箱動作後可以符合引數個數與型態的方法
- 嘗試設有「不定長度引數」並可以符合的方法
- 編譯器找不到合適的方法，回報編譯錯誤

7.2.2 不定長度引數

在呼叫某個方法時，要給方法的引數個數事先無法決定的話該如何處理？例如 System.out.printf() 方法中並沒有辦法事先決定要給的引數個數，像是：

```

System.out.printf("%d", 10);
System.out.printf("%d %d", 10, 20);
System.out.printf("%d %d %d", 10, 20, 30);

```

在 J2SE 5.0 之後開始支援「不定長度引數」（Variable-length Argument），這可以讓您輕鬆的解決這個問題，直接來看範例 7.10 的示範。

範例 7.10 MathTool.java

```

public class MathTool {
    public static int sum(int... nums) { // 使用...宣告參數
        int sum = 0;
        for(int num : nums) {
            sum += num;
        }
        return sum;
    }
}

```

要使用不定長度引數，在宣告參數列時要於型態關鍵字後加上 "...”，而在 sum() 方法的區塊中您可以看到，實際上 nums 是一個陣列，編譯器會將參數列的 (int... nums) 解釋為 (int[] nums)，您可以如範例 7.11 的方式指定各種長度的引數給方法來使用。

範例 7.11 TestVarargs.java

```
public class TestVarargs {
    public static void main(String[] args) {
        int sum = 0;

        sum = MathTool.sum(1, 2);
        System.out.println("1 + 2 = " + sum);

        sum = MathTool.sum(1, 2, 3);
        System.out.println("1 + 2 + 3 = " + sum);

        sum = MathTool.sum(1, 2, 3, 4, 5);
        System.out.println("1 + 2 + 3+ 4+ 5 = " + sum);
    }
}
```

執行結果：

```
1 + 2 = 3
1 + 2 + 3 = 6
1 + 2 + 3+ 4+ 5 = 15
```

編譯器會將傳遞給方法的引數解釋為 int 陣列傳入至 sum() 中，所以實際上不定長度引數的功能也是J2SE 5.0所提供的「編譯蜜糖」（Compiler Sugar）。

在方法上使用不定長度引數時，記得必須宣告的參數必須設定在參數列的最後一個，例如下面的方式是合法的：

```
public void someMethod(int arg1, int arg2, int... varargs) {
    // ...
}
```

但下面的方式是不合法的：

```
public void someMethod(int... varargs, int arg1, int arg2) {
    // ...
}
```

您也沒辦法使用兩個以上的不定長度引數，例如下面的方式是不合法的：

```
public void someMethod(int... varargs1, int... varargs2) {
    // ...
}
```

如果使用物件的不定長度引數，宣告的方法相同，例如：

```
public void someMethod(SomeClass... somes) {
    // ...
}
```

7.2.3 遞迴方法

「遞迴」（Recursion）是在方法中呼叫自身同名方法，而呼叫者本身會先被置入記憶體「堆疊」（Stack）中，等到被呼叫者執行完畢之後，再從堆疊中取出之前被置入的方法繼續執行。堆疊是一種「先進後出」（First in, last out）的資料結構，就好比您將書本置入箱中，最先放入的書會最後才取出。Java 支援遞迴，遞迴的實際應用很多，舉個例子來說，求最大公因數就可以使用遞迴來求解，範例 7.12 是使用遞迴來求解最大公因數的一個實例。

範例 7.12 UseRecursion.java

```
import java.util.Scanner;

public class UseRecursion {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("輸入兩數:");
        System.out.print("m = ");
        int m = scanner.nextInt();

        System.out.print("n = ");
        int n = scanner.nextInt();

        System.out.println("GCD: " + gcd(m, n));
    }

    private static int gcd(int m, int n) {
        if(n == 0)
            return m;
        else
            return gcd(n, m % n);
    }
}
```

執行結果：

```
輸入兩數:
m = 10
n = 20
GCD: 10
```

範例 7.12 是使用輾轉相除法來求最大公因數；遞迴具有重複執行的特性，而可以使用遞迴求解的程式，實際上也可以使用迴圈來求解，例如下面的程式片段就是最大公因數使用迴圈求解的方式。

```
private static int gcd(int m, int n) {
    int r;
    while(n != 0) {
        r = m % n;
        m = n;
        n = r;
    }
    return m;
}
```

使用遞迴好還是使用迴圈求解好？這並沒有一定的答案。由於遞迴本身有重複執行與記憶體堆疊的特性，所以若在求解時需要使用到堆疊特性的資料結構時，使用遞迴在設計時的邏輯會比較容易理解，程式碼設計出來也會比較簡潔，然而遞迴會有方法呼叫的負擔，因而有時會比使用迴圈求解時來得沒有效率，但迴圈求解時若使用到堆疊時，通常在程式碼上會比較複雜。

良葛格的話匣子 在我的網站上有很多題目可以作練習，也不乏有遞迴求解的例子：

- <http://openhome.cc/Gossip/AlgorithmGossip/>

7.2.4 垃圾收集

在解釋「垃圾收集」（Garbage collection）之前，要先稍微提一下 C++ 中對物件資源的管理，以利待會瞭解 Java 的物件資源管理機制。

在 C++ 中，使用 "new" 配置的物件，必須使用 "delete" 來清除物件，以釋放物件所佔據的記憶體空間，如果沒有進行這個動作，若物件不斷的產生，記憶體就會不斷的被物件耗用，最後使得記憶體空間用盡，在 C++ 中有所謂的「解構方法」（Destructor），它會在物件被清除前執行，然而使用 "delete" 並不是那麼的簡單，如果不小心清除了尚在使用中的物件，則程式就會發生錯誤甚至整個崩潰（Crash），如何小心的使用 "new" 與 "delete"，一直是 C++ 中一個重要的課題。

在 Java 中，使用 "new" 配置的物件，基本上也必須清除以回收物件所佔據的記憶體空間，但是您並不用特別關心這個問題，因為 Java 提供垃圾收集機制，在適當的時候，Java 執行環境會自動檢查物件，看看是否有未被參考的物件，如果有的話就清除物件、回收物件所佔據的記憶體空間。

在 Java 中垃圾收集的時機何時開始您並無法得知，可能會在記憶體資源不足的時候，或是在程式執行的空閒時候，您可以建議執行環境進行垃圾收集，但也僅止於建議，如果程式當時有優先權更高的執行緒（Thread）正在進行，則垃圾收集並不一定會馬上進行。

在 Java 中並沒有解構方法，在 Java 中有 finalize() 這個方法，它被宣告為 "protected"，finalize() 會在物件被回收時執行，但您不可以將它當作解構方法來使用，因為不知道物件資源何時被回收，所以也就不知道 finalize() 真正被執行的時間，所以無法立即執行您所指定的資源回收動作，但您可以使用 finalize() 來進行一些相關資源的清除動作，如果這些動作與立即性的收尾動作沒有關係的話。

如果您確定不再使用某個物件，您可以在參考至該物件的名稱上指定 "null"，表示這個名稱不再參考至任何物件，不被任何名稱參考的物件將會被回收資源，您可以使用 System.gc() 建議程式進行垃圾收集，如果建議被採納，則物件資源會被回收，回收前會執行 finalize() 方法。

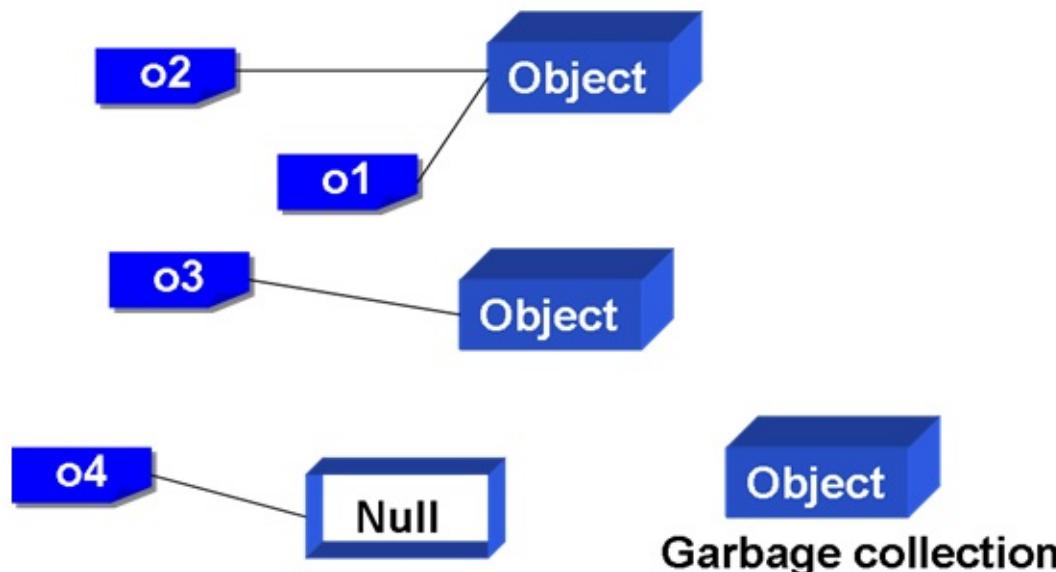


圖 7.5 沒有被名稱參考到的物件資源將會被回收

範例7.13簡單的示範了finalize()方法的使用。 範例7.13 GcTest.java public class GcTest { private String name;

```

public GcTest(String name) {
    this.name = name;
    System.out.println(name + "建立");
}

// 物件回收前執行

```

```
protected void finalize() {
    System.out.println(name + "被回收");
}
```

}.

使用範例 7.14來作個簡單的執行測試。

範例 7.14 UseGC.java

```
public class UseGC {
    public static void main(String[] args) {
        System.out.println("請按Ctrl + C終止程式.....");

        GcTest obj1 = new GcTest("object1");
        GcTest obj2 = new GcTest("object2");
        GcTest obj3 = new GcTest("object3");

        // 令名稱不參考至物件
        obj1 = null;
        obj2 = null;
        obj3 = null;

        // 建議回收物件
        System.gc();

        while(true); // 不斷執行程式
    }
}
```

在程式中您故意加上無窮迴圈，以讓垃圾收集在程式結束前有機會執行，藉以瞭解垃圾收集確實會運作，程式執行結果如下所示：

```
請按Ctrl + C終止程式.....
bject1建立
bject2建立
bject3建立
bject3被回收
bject2被回收
bject1被回收
```

7.3 接下來的主題

每一個章節的內容由淺至深，初學者該掌握的深度要到哪呢？在這個章節中，對於初學者我建議至少掌握以下幾點內容：

- 如何使用 class 定義類別
- 如何定義資料成員
- 如何定義方法成員與建構方法
- 瞭解 this 的作用
- 瞭解靜態 (static) 成員的作用
- 瞭解如何重載方法

在物件導向程式設計中，只是單純的封裝物件特性只能解決一部份的問題，有時候您必須提取出物件的共同抽象特性並加以定義，然後再「繼承」（Inherit）抽象的定義對個別的物件加以實作，有時您必須繼承某個類別並重新改寫類別中的某些定義，這在下一個章節中都會加以說明，並且您也將瞭解「抽象類別」（Abstract class）與「介面」（Interface）的不同。

第 8 章 繼承（Inheritance）、多型（Polymorphism）

在程式設計中，「繼承」（Inheritance）是一把雙面刃，用的好的話可讓整個程式架構具有相當的彈性，用不好的話整個程式會難以維護與修改。「多型」機制本身並不直覺，但使用適當的話可以動態調整物件的呼叫，並降低物件之間的依賴關係。在物件導向的世界中，除了識別出物件並定義類別之外，如何善用「繼承」與「多型」的機制來架構程式，往往都是整個物件導向設計的重心。

這個章節將介紹在 Java 中如何實現繼承與多型，主要側重於語法方面的講解，但會穿插使用繼承與多型時該注意的一些基本觀念與應用，您會瞭解如何擴充類別、如何實作介面，並且認識到在 Java 中，所有物件都是繼承 `java.lang.Object` 類別的事實。

8.1 繼承

您可以基於某個父類別對物件的定義加以擴充，而制訂出一個新的子類別定義，子類別可以繼承父類別原來的某些定義，並也可能增加原來的父類別所沒有的定義，或者是重新定義父類別中的某些特性，事實上，在您開始學會使用 "class" 關鍵字定義類別的同時，您也已經使用了繼承的功能，因為在 Java 中，所有的類別都直接或間接的繼承了 java.lang.Object 類別。

8.1.1 擴充 (extends) 父類別

假設您先前已經為您的動物園遊戲撰寫了一些 Bird 相關類別，現在想要將之擴充，讓動物園擁有像是雞、麻雀等更多鳥的種類，那麼您可以擴充 Bird 類別，這麼一來許多 Bird 中所使用的功能都可以留下來，並基於它擴充一些新的 Chicken、Sparrow 類別，您不用重寫所有的功能，您可以「擴充」（extends）原先已定義好的類別再增加新的定義。

Java 中使用 "extends" 作為其擴充父類別的關鍵字，其實就相當於一般所常稱的「繼承」（Inherit），以動物園中最鳥類為例，假設您原先已定義好一個 Bird 類別，如範例 8.1 所示。

範例 8.1 Bird.java

```
public class Bird {
    private String name;

    public Bird() {
    }

    public Bird(String name) {
        this.name = name;
    }

    public void walk() {
        System.out.println("走路");
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

您可以繼承 Bird 並將之擴充為 Chicken 類別，在擴充（繼承）的關係中，被擴充的類別為「父類別」（Parent class）或「基礎類別」（Base class），而擴充父類別的類別就稱之為「子類別」（Child class）或「衍生類別」（Derived class），父類別跟子類別之間，有「is a」的關係，例如雞「是一種」鳥（Chicken is a bird）。在 Java 中要擴充類別定義時，您要使用 "extends" 關鍵字，並指定要被擴充的類別，如範例 8.2 所示。

範例 8.2 Chicken.java

```
public class Chicken extends Bird { // 擴充Bird類別
    private String crest; // 新增私有成員，雞冠描述

    public Chicken() {
        super();
    }

    // 定義建構方法
    public Chicken(String name, String crest) {
        super(name);
        this.crest = crest;
    }
}
```

```
// 新增方法
public void setCrest(String crest) {
    this.crest = crest;
}

public String getCrest() {
    return crest;
}

public void wu() {
    System.out.println("咕咕叫...");
}

}
```

當您擴充某個類別時，該類別的所有 "public" 成員都可以在衍生類別中被呼叫使用，而 "private" 成員則不可以直接在衍生類別中被呼叫使用；在這個例子中，Bird 類別中已經有 name 成員，您讓 Chicken 擴充 Bird 並新增了 crest 成員，而方法上您新增 "public" 的 getCrest() 等方法，而 getName() 與 walk() 等方法則直接繼承父類別中已定義的內容。

在擴充某個類別之後，您可以一併初始父類別的建構方法，以完成相對應的初始動作，這可以使用 super() 方法來達到，它表示呼叫基底類別的建構方法，super() 必須在建構方法一開始就呼叫，在子類別的建構方法中，如果不用 super() 指定使用父類別中的哪個建構方法來初始，則預設會呼叫父類別中無參數的建構方法。

父類別的 "public" 成員可以直接在衍生類別中使用，而 "private" 成員則不行，"private" 類別只限於定義它的類別之內來存取，如果您想要與父類別的 "private" 成員溝通，就只能透過父類別中繼承下來的 "public" 方法成員，例如範例中的 getName() 方法。

範例 8.3 示範了如何使用擴充了 Bird 的 Chicken 類別。

範例 8.3 ExtendDemo.java

```
public class ExtendDemo {
    public static void main(String[] args) {
        Chicken chicken1 = new Chicken("小克", "紅色小雞冠");
        Chicken chicken2 = new Chicken();

        System.out.printf("小雞1 - 名稱 %s, 雞冠是 %s。 \n",
            chicken1.getName(), chicken1.getCrest());
        chicken1.wu();

        System.out.printf("小雞2 - 名稱 %s, 雞冠是 %s。 \n",
            chicken2.getName(), chicken2.getCrest());
        chicken2.wu();
    }
}
```

Chicken 類別的實例可以直接使用繼承下來的 getName() 方法，並可使用自己的 getCrest()、wu() 方法，執行結果如下所示：

```
小雞1 - 名稱 小克, 雞冠是 紅色小雞冠。
咕咕叫...
小雞2 - 名稱 null, 雞冠是 null。
咕咕叫...
```

良葛格的話匣子 以上是就 Java 的 "extends" 關鍵字之字面意義，只是先對 Java 的繼承作個簡單的介紹，繼承並不只有擴充父類別定義的作用，您可以重新定義（Override）父類別中的方法，或者是將兩個類似的類別行為提取（Pull up）至「抽象類別」（Abstract class）中，將兩個類別歸為同一類，讓它們擁有相同的父類別，這在稍後還會一一介紹。

8.1.2 被保護的（protected）成員

在之前的範例中，資料成員都預設為 "private" 成員，也就是私用成員，私用成員只能在物件內部使用，不能直接透過參考名稱加上 "." 呼叫使用，即使是擴充了該類別的衍生類別也無法直接使用父類別的私用成員，您只能透過父類別所提供的 "public" 方法成員來呼叫或設定私用成員。

然而有些時候，您希望擴充了基底類別的衍生類別，也能夠直接存取基底類別中的成員，而不是透過 "public" 方法成員，但也不是將資料成員宣告為 "public"，因為您只想這些成員被子類別物件所使用，而不希望這些成員被其它外部物件直接呼叫使用。

您可以宣告這些成員為「被保護的」（protected）成員，保護的意思表示存取該成員是有條件限制的，當您將類別的成員宣告為受保護的成員之後，繼承的類別就可以直接使用這些成員，但這些成員仍然受到保護，不同套件（package）的物件不可直接呼叫使用 protected 成員（關於套件的說明，請看第 9 章）。

要宣告一個成員成為受保護的成員，就使用 "protected" 關鍵字，範例 8.4 是個實際的例子，您將資料成員宣告為受保護的成員。

範例 8.4 Rectangle.java

```
public class Rectangle {
    // 受保護的member
    protected int x;
    protected int y;
    protected int width;
    protected int height;

    public Rectangle() {
    }

    public Rectangle(int x, int y,
                    int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }

    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }
    public void setWidth(int width) { this.width = width; }
    public void setHeight(int height) { this.height = height; }

    public int getX() { return x; }
    public int getY() { return y; }
    public int getWidth() { return width; }
    public int getHeight() { return height; }

    public int getArea() { return width*height; }
}
```

成員如果被宣告為 "protected"，則擴充它的子類別就可以直接使用該資料成員，而不用透過 "public" 方法成員來呼叫，如範例 8.5 所示範的。

範例 8.5 Cubic.java

```
public class Cubic extends Rectangle {
    protected int z;
    protected int length;

    public Cubic() {
        super();
    }

    public Cubic(int x, int y, int z,
                int length, int width, int height) {
        super(x, y, width, height);
```

```

        this.z = z;
        this.length = length;
    }

    public void setZ(int z) { this.z = z; }
    public void setLength(int length) { this.length = length; }

    public int getZ() { return z; }
    public int getLength() { return length; }

    public int getColumn() {
        // 可以直接使用父類別中的width、height成員
        return length*width*height;
    }
}

```

可以直接使用繼承下來的受保護成員確實比較方便，方法成員也可以宣告為受保護的成員。父類別中想要讓子類別擁有的資料成員會宣告為 "protected"，父類別中想要子類別也可以使用的工具方法也會宣告為 "protected"，這些方法對不同套件的物件來說，可能是呼叫它並沒有意義或是有權限上的危險性，但您在衍生類別中仍可能使用到這些方法。

良葛格的話匣子 在設計類別的過程中，並不是一開始就決定哪些資料成員或方法要設定為 "protected"，通常資料成員或非公開的方法都會先宣告為 "private"，當物件的職責擴大而要開始使用繼承時，再逐步考量哪些成員要設定為 "protected"。

8.1.3 重新定義（Override）方法

類別是物件的定義書，如果父類別中的定義並不符合您的需求，可以在擴充類別的同時重新定義，您可以重新定義方法的實作內容、成員的存取權限，或是成員的返回值型態（重新定義返回值型態是 J2SE 5.0 新功能）。

舉個例子來說，看看下面這個類別：

```

public class SimpleArray {
    protected int[] array;

    public SimpleArray(int i) {
        array = new int[i];
    }
    public void setElement(int i, int data) {
        array[i] = data;
    }
    ...
}

```

這個類別設計一個簡單的陣列工具類別，不過您覺得它的 `setElement()` 方法不夠安全，您希望增加一些陣列的長度檢查動作，於是您擴充該類別並重新定義 `setElement()` 方法：

```

public class SafeArray extends SimpleArray {
    public SafeArray(int i) {
        super(i);
    }
    // 重新定義setElement()
    public void setElement(int i, int data) {
        if(i < array.length)
            super.setElement(i, data);
    }
    ...
}

```

這麼一來，以 `SafeArray` 類別的定義所產生的物件，就可以使用新的定義方法，就 `SafeArray` 類別來說，由於操作介面與 `SimpleArray` 是一致的，所以您可以這麼使用：

```
SimpleArray simpleArray = new SafeArray();
simpleArray.setElement();
```

SafeArray 與 SimpleArray 擁有一致的操作介面，因為 SafeArray 是 SimpleArray 型態的子類，擁有從父類中繼承下來的 setElement() 操作介面，雖然您使用 SimpleArray 型態的介面來操作 SafeArray 的實例，但由於實際運作的物件是 SafeArray 的實例，所以被呼叫執行的會是 SafeArray 中重新定義過的 setElement() 方法，這是「多型」（Polymorphism）操作的一個例子，8.2 中對多型操作還會再作說明。

在上頁的程式片段中您看到 super() 與 super 的使用，如果您在衍生類別中想要呼叫基底類別的建構方法，可以使用 super() 方法；若您要在衍生類別中呼叫基底類別方法，則可以如使用 super.methodName()，但使用 super() 呼叫父類別建構方法或使用 super.methodName() 呼叫父類別中方法是有條件的，也就是父類別中的方法或建構方法不能是 "private"，也就是不能是私用成員。

重新定義方法時要注意的是，您可以增大父類別中的方法權限，但不可以縮小父類別的方法權限，也就是若原來成員是 "public" 的話，您不可以在父類別中重新定義它為 "private" 或 "protected"，所以在擴充 SimpleArray 時，您就不可以這麼作：

```
public class SafeArray extends SimpleArray {
    // 不可以縮小父類別中同名方法的權限
    private void setElement(int i, int data) {
        ...
    }
}
```

嘗試將 setElement() 方法從 "public" 權限縮小至 "private" 權限是不行的，在進行編譯時編譯器會回報以下的錯誤訊息：

```
elementType(int,int) in SafeArray cannot override elementType(int,int) in SimpleArray;      attempting to assign weaker access privileges to an inherited element
private void setElement(int i, int data) {
^
1 error
```

從 J2SE 5.0 開始在重新定義方法時，您可以重新定義返回值的型態，例如您原先設計了一個 Bird 類別：

```
public class Bird {
    protected String name;

    public Bird(String name) {
        this.name = name;
    }
    public Bird getCopied {
        return new Bird(name);
    }
}
```

getCopied() 方法原來返回的是Bird物件，現在打算擴充Bird類別，您繼承它並定義了一個 Chicken 類別，在 J2SE 5.0 之前，您會很苦惱於不能重新定義返回值型態，因此您勢必要重新寫一個方法名稱來傳回 Chicken 型態的返回值，但是從 J2SE 5.0 開始，重新定義返回值型態是可行的，重新定義返回值型態有限制條件，重新定義的返回值型態必須是父類別中同一方法返回型態的子類別，例如 Chicken 可以這麼定義：

```
public class Chicken extends Bird {
    protected String crest;

    public Point3D(String name, String crest) {
        super(name);
        this.crest = crest;
```

```

    }
    // 重新定義返回值型態為Point3D
    public Chicken getCopied() {
        return new Chicken(name, crest);
    }
}

```

在 getCopied() 方法的返回值型態上，父類別中返回的是Bird型態，子類別在重新定義 getCopied() 方法時，可以重新定義一個 Bird 型態的子類別型態之返回值，在上面的程式片段中，所重新定義的返回值型態是 Chicken，它是 Bird 的子類別。

注意！您無法重新定義 static 方法，一個方法要被重新定義，它必須是非 static 的，如果您在子類別中定義一個有同樣簽署 (signature) 的 static 成員，那不是重新定義，那是定義一個屬於該子類別的 static 成員。

8.1.4 Object 類別

在 Java 中只要使用 "class" 關鍵字定義類別，您就開始使用繼承的機制了，因為在 Java 中所有的物件都擴充自 java.lang.Object 類別，Object 類別是 Java 程式中所有類別的父類別，每個類別都直接或間接繼承自 Object 類別，當您如下定義一個類別時：

```

public class Foo {
    // 實作
}

```

在 Java 中定義類別時如果沒有指定要繼承的類別，則自動繼承 Object 類別，上面的程式片段即等於您如下定義類別：

```

public class Foo extends Object {
    // 實作
}

```

由於 Object 類別是 Java 中所有類別的父類別，所以使用 Object 壓告的名稱，可以參考至任何的物件而不會發生任何錯誤，因為每一個物件都是 Object 的子物件，舉個簡單的例子，您可以製作一個簡單的集合 (Collection) 類別，並將一些自訂的類別之實例加入其中，範例 8.6 是個簡單示範。

範例 8.6 SimpleCollection.java

```

public class SimpleCollection {
    private Object[] objArr;
    private int index = 0;

    public SimpleCollection() {
        // 預設10個物件空間
        objArr = new Object[10];
    }

    public SimpleCollection(int capacity) {
        objArr = new Object[capacity];
    }

    public void add(Object o) {
        objArr[index] = o;
        index++;
    }

    public int getLength() {
        return index;
    }

    public Object get(int i) {
        return objArr[i];
    }
}

```

接著您如範例 8.7、範例 8.8 定義兩個簡單的類別。

範例 8.7 Foo1.java

```
public class Foo1 {
    private String name;

    public Foo1(String name) {
        this.name = name;
    }

    public void showName() {
        System.out.println("foo1 名稱：" + name);
    }
}
```

範例 8.8 Foo2.java

```
public class Foo2 {
    private String name;

    public Foo2(String name) {
        this.name = name;
    }

    public void showName() {
        System.out.println("foo2 名稱：" + name);
    }
}
```

範例 8.6 的 SimpleCollection 以 Object 陣列來儲存物件，您可以撰寫一個簡單的測試類別來執行看看。

範例 8.9 SimpleCollectionDemo.java

```
public class SimpleCollectionDemo {
    public static void main(String[] args) {
        SimpleCollection simpleCollection =
            new SimpleCollection();

        simpleCollection.add(new Foo1("一號 Foo1"));
        simpleCollection.add(new Foo2("一號 Foo2"));

        Foo1 f1 = (Foo1) simpleCollection.get(0);
        f1.showName();

        Foo2 f2 = (Foo2) simpleCollection.get(1);
        f2.showName();
    }
}
```

在程式中，SimpleCollection 物件可以加入任何型態的物件至其中，因為所有的物件都是 Object 的子物件，從 SimpleCollection 指定索引取回物件時，您要將物件的類型從 Object 轉換為原來的類型，如此就可以操作物件上的方法，執行結果如下：

```
foo1 名稱：一號 Foo1
foo2 名稱：一號 Foo2
```

事實上，如果您需要某種型式的物件容器，Java SE 中就已經有提供了，像是 java.util.List、java.util.Map、java.util.Set 等，

您可以先看看 Java SE 所提供的種種容器類別是不是符合您的需求，關於這些容器類別，在第 13 章中也會介紹。

Object 類別定義了幾個方法，包括 "protected" 權限的 clone()、 finalize()，以及 "public" 權限的 equals()、 toString()、 hashCode()、 notify()、 notifyAll()、 wait()、 getClass() 等方法。除了 getClass()、 notify()、 notifyAll()、 wait() 等方法之外，其它方法您都可以在繼承之後加以重新定義（因為 getClass()、 notify()、 notifyAll()、 wait() 等方法被宣告為 "final"，所以無法重新定義，關於 "final"，8.1.7 就會介紹），以符合您所建立的類別需求。

finalize() 已經在 7.2.4 介紹過了；clone() 用於物件複製，稍後即會介紹；notify()、 notifyAll()、 wait() 是有關於執行緒（Thread）操作，會在第 15 章介紹；getClass() 可以取得類別的 Class 實例，會在第 16 章介紹它的用途；以下先介紹 toString()、 equals()、 hashCode() 方法的使用。

8.1.5 **toString()、 equals()、 hashCode()** 方法

Object 的 toString() 方法是對物件的文字描述，它會返回 String 實例，您在定義物件之後可以重新定義 toString() 方法，為您的物件提供特定的文字描述，Object 的 toString() 預設會傳回類別名稱及 16 進位制的編碼，也就是傳回以下的字串：

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

getClass() 方法是 Object 中定義的方法，它會傳回物件於執行時期的 Class 實例，再使用 Class 實例的 getName() 方法可以取得類別名稱；hashCode() 傳回該物件的「雜湊碼」（Hash code）。Object 的 toString() 方法預設在某些場合是有用的，例如 StringBuilder 就改寫了 toString() 方法，您可以呼叫 StringBuilder 實例的 toString() 方法以得到建構好的 String 實例（6.1.3 曾經介紹過 StringBuilder），範例 8.10 是個簡單的示範。

範例 8.10 ToStringDemo.java

```
public class ToStringDemo {
    public static void main(String[] args) {
        StringBuilder builder = new StringBuilder();

        for(int i = 0; i < 10; i++)
            builder.append(i);

        System.out.println(builder.toString());
    }
}
```

執行結果：

```
0123456789
```

良葛格的話匣子「雜湊碼」（Hash code）由雜湊函數計算得到，在資料結構上可用於資料的定址，Java 中的 java.util.HashMap（第 13 章會介紹）也是利用雜湊的原理來快速儲存與尋找物件，雖然在 Java 中您不用瞭解雜湊的原理，也可以直接使用像 HashMap 類別以得到使用雜湊的好處，但有機會的話，仍建議您看看資料結構中有關於雜湊的原理介紹。

接著介紹 equals() 與 hashCode() 方法，Object 預設的 equals() 本身是比較物件的記憶體位址是否相同，您可以重新定義 equals() 方法，以定義您自己的物件在什麼條件下可視為相等的物件，然而在重新定義 equals() 方法時，建議同時重新定義 hashCode() 方法，因為在以雜湊（hash）為基礎的相關環境中，需要比較兩個物件是否為相同的物件時，除了使用 equals() 之外，還會依 hashCode() 方法來作判斷，例如加入 java.util.HashSet 容器（第 13 章中會介紹）中的物件就必須重新定義 equals() 與 hashCode() 方法，以作為加入 HashSet 中唯一物件的識別。

來看看重新定義 equals() 與 hashCode() 方法的實際例子，您可以根據物件中真正包括的資料成員來比較兩個物件是否相同，如範例 8.11 所示範的。

範例 8.11 Cat.java

```

import java.util.Date;

public class Cat {
    private String name;
    private Date birthday;

    public Cat() {}

    public void setName(String name) { this.name = name; }
    public String getName() { return name; }

    public void setBirthday(Date birthday) { this.birthday = birthday; }
    public Date getBirthday() { return birthday; }

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

        if (!(other instanceof Cat))
            return false;

        final Cat cat = (Cat) other;

        if (!getName().equals(cat.getName()))
            return false;

        if (!getBirthday().equals(cat.getBirthday()))
            return false;

        return true;
    }

    public int hashCode() {
        int result = getName().hashCode();
        result = 29 * result + getBirthday().hashCode();
        return result;
    }
}

```

這是一個根據「商務鍵值」（Business key）實作 equals() 與 hashCode() 的例子，實際上開始實作時要根據您實際的需求，決定如何利用相關的商務鍵值來組合以重新定義這兩個方法。

8.1.6 clone() 方法

clone() 方法是有關於如何複製物件本身的方法，您可以重新定義您的複製方法，不過物件的複製要深入的話必須考慮很多細節，因為繼承類別、實作介面、物件依賴等重重的關係，會使得要完整複製一個物件的資訊變得困難，在以下我將介紹一個最基本的作法：實作 java.lang.Cloneable 介面（Interface）。

關於介面會在下一節中詳細介紹，然而因為 Cloneable 介面沒有定義任何的方法，所以目前您只需要先知道，要進行物件複製，您需要在定義類別上掛上一個識別介面，表示這個類別的實例支援自身複製。直接以實際的例子來說明，首先您可以定義一個 Point 類別，並讓這個類別的實例可以複製自己，如範例 8.12 所示。

範例 8.12 Point.java

```

public class Point implements Cloneable { // 要實作Cloneable
    private int x;
    private int y;

    public Point() {}
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

```

```

public void setX(int x) { this.x = x; }
public void setY(int y) { this.y = y; }

public int getX() { return x; }
public int getY() { return y; }

public Object clone() throws CloneNotSupportedException {
    // 呼叫父類別的clone()來進行複製
    return super.clone();
}
}

```

接著再定義一個 Table 類別，當中定義了 Point 為它的資料成員之一，當您複製 Table 的實例時，理所當然的也要複製資料成員，範例 8.13 的程式碼示範了如何定義 clone() 方法。

範例 8.13 Table.java

```

public class Table implements Cloneable { // 要實作Cloneable
    private Point center;

    public void setCenter(Point center) {
        this.center = center;
    }
    public Point getCenter() {
        return center;
    }

    public Object clone ()
        throws CloneNotSupportedException {
        // 呼叫父類的clone()來複製
        Table table = (Table) super.clone();

        if(this.center != null) {
            // 複製Point類型的資料成員
            table.center = (Point) center.clone();
        }

        return table;
    }
}

```

注意 Point 類別與 Table 類別都實作了 Cloneable 介面，要支援複製自身的物件，其定義類別時必須實作 Cloneable 介面，如果不實作這個介面的類別，其實例的 clone() 方法被呼叫時，會丟出 CloneNotSupportedException 例外，Java 中的 clone() 方法是繼承自 Object，為了方便直接呼叫，您將其存取權限從 "protected" 改為 "public"（您也可以再定義一個"public"方法來傳回複製的物件，這邊改成 "public" 只是為了方便，可以少寫幾行程式碼）。

接著來寫一個簡單的測試類別，範例 8.14 的 Table 實例會被複製，您可以改變複製品的內容，但原作品的內容並不會受到影響，表示兩個實例是不相互干涉的。

範例 8.14 CloneDemo.java

```

public class CloneDemo {
    public static void main(String[] args)
        throws CloneNotSupportedException {
        Table table = new Table();
        table.setCenter(new Point(2, 3));
        Point originalCenter = table.getCenter();

        Table clonedTable = (Table) table.clone();
        Point clonedCenter = clonedTable.getCenter();

        System.out.printf("原來的Table中心 : (%d, %d)\n",
            originalCenter.getX(), originalCenter.getY());
        System.out.printf("複製的Table中心 : (%d, %d)\n",
            clonedCenter.getX(), clonedCenter.getY());
    }
}

```

```

clonedCenter.setX(10);
clonedCenter.setY(10);

// 改變複製品的內容，對原來的物件不會有影響
System.out.printf("原來的Table中心 : (%d, %d)\n",
    originalCenter.getX(), originalCenter.getY());
System.out.printf("複製的Table中心 : (%d, %d)\n",
    clonedCenter.getX(), clonedCenter.getY());
}
}

```

執行結果如下：

```

原來的Table中心 : (2, 3)
複製的Table中心 : (2, 3)
原來的Table中心 : (2, 3)
複製的Table中心 : (10, 10)

```

8.1.7 final 關鍵字

"final" 關鍵字可以使用在變數宣告時，表示該變數一旦設定之後，就不可以再改變該變數的值，例如在下面的程式碼片段中，PI 這個變數一旦設定，就不可以再有指定值給 PI 的動作：

```
final double PI = 3.14159;
```

如果在定義方法成員時使用 "final"，表示該方法成員在無法被子類別重新定義（Override），例如：

```

public class Ball {
    private double radius;

    public final double getRadius() {
        return radius;
    }
    // ...
}

```

您在繼承 Ball 類別後，由於 getRadius() 方法被宣告為 "final"，所以在子類別中 getRadius() 方法不能再被重新定義。如果您在宣告類別時加上 final 關鍵字，則表示要終止被擴充，這個類別不可以被其它類別繼承，例如：

```

public final class Ball {
    // ...
}

```

8.2 多型 (Polymorphism)

多型操作指的是使用同一個操作介面，以操作不同的物件實例，多型操作在物件導向上是為了降低對操作介面的依賴程度，進而增加程式架構的彈性與可維護性。多型操作是物件導向上一個重要的特性，這個小節會介紹多型的觀念，以及「抽象類別」（Abstract）與「介面」（Interface）應用的幾個實例。

8.2.1 多型導論

先來解釋一下這句話：多型操作指的是使用同一個操作介面，以操作不同的物件實例。首先來解釋一下何謂操作介面，就 Java 程式而言，操作介面通常指的就是您在類別上定義的公開方法，透過這些介面，您可以對物件實例加以操作。

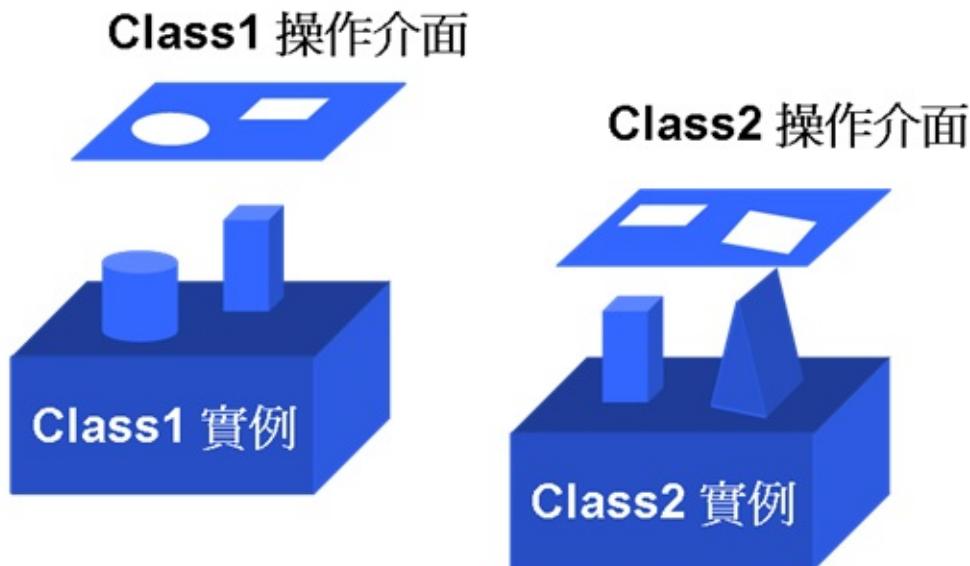


圖 8.1 透過對應介面來操作物件

圖 8.1 中上方的平面相當於類別定義的介面，方塊上凸出的部份相當於物件實例上可操作的方法，要操作物件上的方法必須使用對應型態的操作介面，而如果您使用不正確的類別型態來轉換物件的操作介面，則會發生 `java.lang.ClassCastException` 例外，例如：

```
Class1 c1 = new Class1();
Class2 c2 = (Class2) c1; // 丟出ClassCastException例外
```

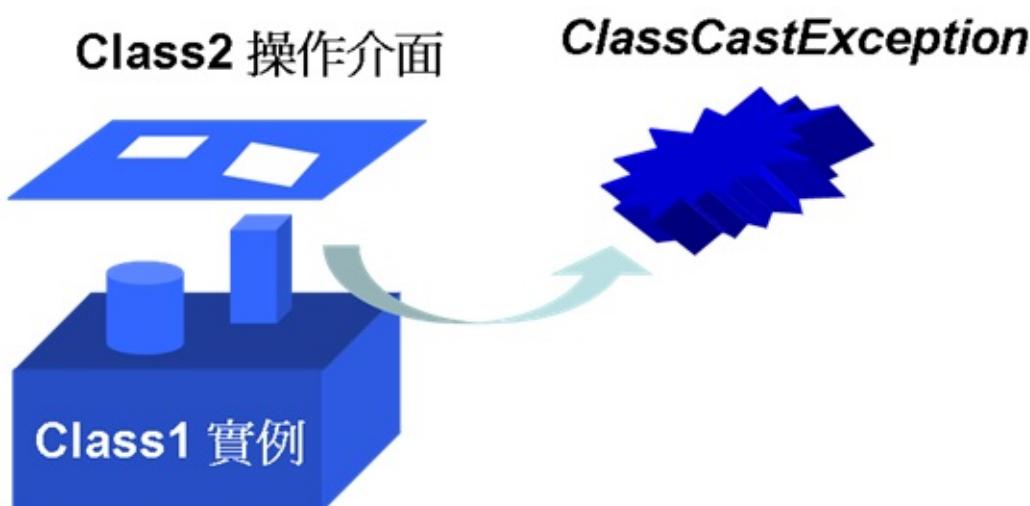


圖 8.2 不正確的型態轉換會丟出 ClassCastException 例外

回到多型操作的解釋上，現在假設 Class1 上定義了 doSomething() 方法，而 Class2 上也定義了 doSomething() 方法，而您定義了兩個 execute() 方法來分別操作 Class1 與 Class2 的實例：

```
public void execute(Class1 c1) {
    c1.doSomething();
}
public void execute(Class2 c2) {
    c2.doSomething();
}
```

很顯然的，您的程式中 execute() 分別依賴了 Class1 與 Class2 兩個類別，與其依賴兩個類別，不如定義一個父類別 ParentClass 類別，當中定義有 doSomething()，並讓 Class1 與 Class2 都繼承 ParentClass 類別並重新定義自己的 doSomething() 方法，如此您就可以將程式中的 execute() 改成：

```
public void execute(ParentClass c) {
    c.doSomething();
}
```

這是可以行的通的，因為介面與實例上的操作方法是一致的，如圖 8.3 所示。

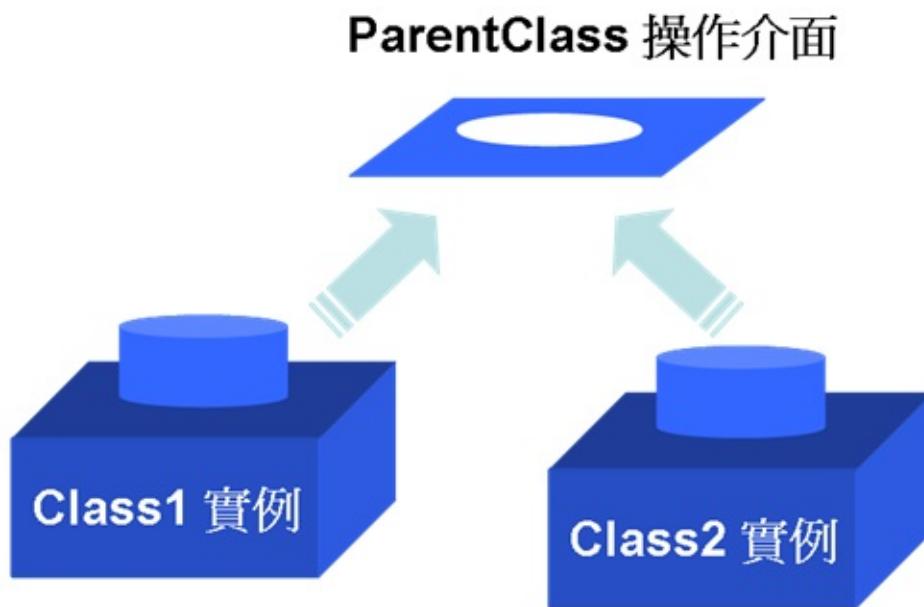


圖 8.3 Class1 與 Class2 是 ParentClass 的子類，可以透過 ParentClass 來操作

這就是多型操作所指的，使用同一個操作介面，以操作不同的物件實例。由於從分別依賴 Class1 與 Class2 改為只依賴 ParentClass，程式對個別物件的依賴程度降低了，日後在修改、維護或調整程式時的彈性也增加了，這是繼承上多型操作的一個實例。

以上是對多型的一個簡介，實際上在設計並不依賴於具體類別，而是依賴於抽象，Java 中在實現多型時，可以讓程式依賴於「抽象類別」（Abstract class）或是「介面」（Interface），雖然兩者都可以實現多型操作，但實際上兩者的語義與應用場合是不同的，接下來我將分別介紹兩者的使用方式與應用實例。

8.2.2 抽象類別（Abstract class）

在 Java 中定義類別時，可以僅宣告方法名稱而不實作當中的邏輯，這樣的方法稱之為「抽象方法」（Abstract method），如果一個方法中包括了抽象方法，則該類別稱之為「抽象類別」（Abstract class），抽象類別是擁有未實作方法的類別，所

以它不能被用來生成物件，它只能被繼承擴充，並於繼承後實作未完成的抽象方法，在 Java 中要宣告抽象方法與抽象類別，您要使用 "abstract" 關鍵字，以下舉個實際的例子，先假設您設計了兩個類別：ConcreteCircle 與 HollowCircle。

```
public class ConcreteCircle {
    private double radius;
    public void setRadius(int radius) { this.radius = radius; }
    public double getRadius() { return radius; }
    public void render() {
        System.out.printf("畫一個半徑 %f 的實心圓\n", getRadius());
    }
}

public class HollowCircle {
    private double radius;
    public void setRadius(int radius) { this.radius = radius; }
    public double getRadius() { return radius; }
    public void render() {
        System.out.printf("畫一個半徑 %f 的空心圓\n", getRadius());
    }
}
```

顯然的，這兩個類別除了 render() 方法的實作內容不同之外，其它的定義是一樣的，而且這兩個類別所定義的顯然都是「圓」的一種類型，您可以定義一個抽象的 AbstractCircle 類別，將 ConcreteCircle 與 HollowCircle 中相同的行為與定義提取（Pull up）至抽象類別中，如範例 8.15 所示。

範例 8.15 AbstractCircle.java

```
public abstract class AbstractCircle {
    protected double radius;

    public void setRadius(int radius) { this.radius = radius; }
    public double getRadius() { return radius; }

    public abstract void render();
}
```

注意到在類別宣告上使用了 "abstract" 關鍵字，所以 AbstractCircle 是個抽象類別，它只能被繼承，而 render() 方法上也使用了 "abstract" 關鍵字，表示它是個抽象方法，目前還不用實作這個方法，繼承了 AbstractCircle 的類別必須實作 render() 方法，接著您可以讓 ConcreteCircle 與 HollowCircle 類別繼承 AbstractCircle 方法並實作 render() 方法，如範例 8.16、範例 8.17 所示範的。

範例 8.16 ConcreteCircle.java

```
public class ConcreteCircle extends AbstractCircle {
    public ConcreteCircle() {}

    public ConcreteCircle(double radius) {
        this.radius = radius;
    }

    public void render() {
        System.out.printf("畫一個半徑 %f 的實心圓\n", getRadius());
    }
}
```

範例 8.17 HollowCircle.java

```
public class HollowCircle extends AbstractCircle {
    public HollowCircle() {}
```

```

public HollowCircle(double radius) {
    this.radius = radius;
}

public void render() {
    System.out.printf("畫一個半徑 %f 的空心圓\n", getRadius());
}
}

```

由於共同的定義被提取至 AbstractCircle 類別中，並於擴充時繼承了下來，所以在 ConcreteCircle 與 HollowCircle 中不用重複定義，只要定義個別對 render() 的處理方式就行了，而由於 ConcreteCircle 與 HollowCircle 都是 AbstractCircle 的子類別，因而可以使用 AbstractCircle 上有定義的操作介面，來操作子類別實例上的方法，如範例 8.18 所示範的。

範例 8.18 CircleDemo.java

```

public class CircleDemo {
    public static void main(String[] args) {
        renderCircle(new ConcreteCircle(3.33));
        renderCircle(new HollowCircle(10.2));
    }

    public static void renderCircle(AbstractCircle circle) {
        circle.render();
    }
}

```

由於 AbstractCircle 上有定義 render() 方法，所以可用於操作子類別實例的方法，這是繼承上多型操作的一個實例應用，執行結果如下所示：

```

畫一個半徑 3.330000 的實心圓
畫一個半徑 10.200000 的空心圓

```

以上所舉的例子是 8.2.1 多型導論的具體例子，對 renderCircle() 方法來說，它只需依賴 AbstractCircle 類別，而不用個別為 ConcreteCircle 與 HollowCircle 類別撰寫個別的 renderCircle() 方法。

良葛格的話匣子 將抽象類別的名稱加上 Abstract 作為開頭，可表明這是個抽象類別，用意在提醒開發人員不要使用這個類別來產生實例（事實上也無法產生實例）。

在程式撰寫的過程中會像這邊所介紹的，將已有的程式加以「重構」（Refactor），讓物件職責與程式架構更有彈性，事實上這邊的例子就使用了重構中的「Pull up field」與「Pull up method」方法，重構手法是程式開發的經驗集成，如果您對這些經驗有興趣，建議您看看這本書：

Refactoring: Improving the Design of Existing Code by Martin Fowler, Kent Beck, John Brant, William Opdyke, don Roberts

8.2.3 抽象類別應用

為了加深您對抽象類別的瞭解與應用方式，再來舉一個例子說明抽象類別，在範例 8.19 中定義了一個簡單的比大小遊戲抽象類別。

範例 8.19 AbstractGuessGame.java

```

public abstract class AbstractGuessGame {
    private int number;

    public void setNumber(int number) {
        this.number = number;
    }
}

```

```

    }

    public void start() {
        showMessage("歡迎");
        int guess = 0;
        do {
            guess = getUserInput();
            if(guess > number) {
                showMessage("輸入的數字較大");
            }
            else if(guess < number) {
                showMessage("輸入的數字較小");
            }
            else {
                showMessage("猜中了");
            }
        } while(guess != number);
    }

    protected abstract void showMessage(String message);
    protected abstract int getUserInput();
}

```

這是個抽象類別，您在類別定義了 start() 方法，當中先實作比大小遊戲的基本規則，然而並不實作如何取得使用者輸入及訊息的顯示方式，只先定義了抽象方法 showMessage() 與 getUserInput()，使用 AbstractGuessGame 類別的辦法是擴充它，並實作當中的抽象方法，例如您可以實作一個簡單的文字介面遊戲類別，如範例 8.20 所示。

範例 8.20 TextModeGame.java

```

import java.util.Scanner;

public class TextModeGame extends AbstractGuessGame {
    private Scanner scanner;

    public TextModeGame() {
        scanner = new Scanner(System.in);
    }

    protected void showMessage(String message) {
        for(int i = 0; i < message.length()*2; i++) {
            System.out.print("**");
        }
        System.out.println("\n"+ message);
        for(int i = 0; i < message.length()*2; i++) {
            System.out.print("**");
        }
    }

    protected int getUserInput() {
        System.out.print("\n輸入數字 : ");
        return scanner.nextInt();
    }
}

```

範例 8.21 是啟動遊戲的示範類別。

範例 8.21 GuessGameDemo.java

```

public class GuessGameDemo {
    public static void main(String[] args) {
        AbstractGuessGame guessGame =
            new TextModeGame();
        guessGame.setNumber(50);
        guessGame.start();
    }
}

```

執行結果：

```
*****
歡迎
*****
輸入數字：10
*****
輸入的數字較小
*****
輸入數字：50
*****
猜中了
*****
```

如果您想要實作一個有視窗介面的比大小遊戲，則您可以擴充 AbstractGuessGame 並在抽象方法 showMessage() 與 getUserInput() 中實作有視窗介面的訊息顯示；藉由在抽象類別中先定義好程式的執行流程，並將某些相依方法留待子類別中執行，這是抽象類別的應用場合之一。

良葛格的話匣子 事實上這邊的例子是「Template Method 模式」的一個實例，Template Method 模式是Gof (Gang of Four) 設計模式 (Design Pattern) 名書中23種模式中的一個，建議您在具備基本的物件導向觀念之後看看設計模式的相關書籍，可以增加您在物件導向程式設計上的功力，Gof 設計模式書是：

Design Patterns Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

我的網站上也整理有一些設計模式相關資料，並附有簡單的Java程式實例，您也可以一併參考：

- <http://openhome.cc/Gossip/DesignPattern/>

8.2.4 介面 (Interface)

表面上看來，介面有點像是完全沒有任何方法被實作的抽象類別，但實際上兩者在語義與應用上是有差別的。「繼承某抽象類別的類別必定是該抽象類別的一個子類」，由於同屬一個類型，只要父類別中有定義同名方法，您就可以透過父類別型態來操作子類實例中被重新定義的方法，也就是透過父類別型態進行多型操作，但「實作某介面的類別並不被歸屬於哪一類」，一個物件上可以實作多個介面。

考慮您有一個方法 doRequest()，您事先並無法知道什麼型態的物件會被傳進來，或者是這個方法可以接受任何類型的物件，您想要操作物件上的某個特定方法，例如 doSomething() 方法，問題是傳進來的物件是任意的，除非您定義一個抽象類別並宣告 doSomething() 抽象方法，然後讓所有的類別都繼承這個抽象類別，否則的話您的 doRequest() 方法似乎無法實作出來，實際上這麼作也沒有價值。

介面的目的在定義一組可操作的方法，實作某介面的類別必須實作該介面所定義的所有方法，只要物件有實作某個介面，就可以透過該介面來操作物件上對應的方法，無論該物件實際上屬於哪一個類別，像上面所述及的問題，就要靠要介面來解決。

介面的宣告是使用 "interface" 關鍵字，宣告方式如下：

```
[public] interface 介面名稱 {
    權限設定 傳回型態 方法(參數列);
    權限設定 傳回型態 方法(參數列);
    // ...
}
```

在宣告介面時方法上的權限設定可以省略，如果省略的話，預設是 "public"，來看宣告介面的一個實例。

範例 8.22 IRequest.java

```
public interface IRequest {
    public void execute();
}
```

在定義類別時，您可以使用“implements”關鍵字來指定要實作哪個介面，介面中所有定義的方法都要實作，範例 8.23、範例 8.24 都實作了範例 8.22 的 IRequest 介面。

範例 8.23 HelloRequest.java

```
public class HelloRequest implements IRequest {
    private String name;

    public HelloRequest(String name) {
        this.name = name;
    }

    public void execute() {
        System.out.printf("哈囉 %s ! %n", name);
    }
}
```

範例 8.24 WelcomeRequest.java

```
public class WelcomeRequest implements IRequest {
    private String place;

    public WelcomeRequest(String place) {
        this.place = place;
    }

    public void execute() {
        System.out.printf("歡迎來到 %s ! %n", place);
    }
}
```

假設您設計了一個 doRequest()方法，雖然 HelloRequest 與 WelcomeRequest 是兩種不同的類型（類別），但它們都實現了 IRequest，所以 doRequest() 只要知道 IRequest 定義了什麼方法，就可以操作 HelloRequest 與 WelcomeRequest 的實例，而不用知道傳入的物件到底是什麼類別的實例，範例 8.25 是這個觀念的簡單示範。

範例 8.25 RequestDemo.java

```
public class RequestDemo {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++) {
            int n = (int) (Math.random() * 10) % 2; // 隨機產生
            switch (n) {
                case 0:
                    doRequest(new HelloRequest("良葛格"));
                    break;
                case 1:
                    doRequest(new WelcomeRequest("Wiki 網站"));
            }
        }
    }

    public static void doRequest(IRequest request) {
        request.execute();
    }
}
```

在範例 8.25 中傳遞給 doRequest() 的物件是隨機的，雖然實際上傳入的物件並不知道到底是HelloRequest的實例，或者是WelcomeRequest的實例，但 doRequest() 知道傳入的物件都有實作 IRequest 介面上的方法，所以執行時就按照 IRequest 定義的方法來操作物件，執行結果如下：

```
哈囉 良葛格！
哈囉 良葛格！
哈囉 良葛格！
歡迎來到 Wiki 網站！
哈囉 良葛格！
哈囉 良葛格！
歡迎來到 Wiki 網站！
哈囉 良葛格！
歡迎來到 Wiki 網站！
歡迎來到 Wiki 網站！
```

在 Java 中您可以一次實作多個介面，實作多個介面的方式如下：

```
public class 類別名稱 implements 介面1, 介面2, 介面3 {
    // 介面實作
}
```

當您實作多個介面時，記得必須實作每一個介面中所定義的方法，由於實作了多個介面，所以要操作物件時，必要時必須作「介面轉換」，如此程式才知道如何正確的操作物件，假設 someObject 實作了 ISomeInterface1 與 ISomeInterface2 兩個介面，則您可以如下對物件進行介面轉換與操作：

```
ISomeInterface1 obj1 = (ISomeInterface1) someObject;
obj1.doSomeMethodOfISomeInterface1();

ISomeInterface2 obj2 = (ISomeInterface2) someObject;
obj2.doSomeMethodOfISomeInterface2();
```

簡單的說，您每多實作一個介面，就要多遵守一個實作協議。介面也可以進行繼承的動作，同樣也是使用 "extends" 關鍵字來繼承父介面，例如：

```
public interface 名稱 extends 介面1, 介面2 {
    // ...
}
```

不同於類別一次只能繼承一個父類別，一個介面可以同時繼承多個父介面，實作子介面的類別必須將所有在父介面和子介面中定義的方法實作出來。

良葛格的話匣子 在定義介面名稱時，可以使用 'I' 作為開頭，例如 IRequest 這樣的名稱，表明它是一個介面（Interface）。

事實上範例 8.25 是「Command 模式」的一個簡化實例，同樣的也可以參考 Gof 的設計模式書籍，我的網站上也有 Command 模式的介紹。

在設計上鼓勵依賴關係儘量發生在介面上，雖然抽象類別也可以達到多型操作的目的，但依賴於抽象類別，表示您也依賴於某個類型（類別），而依賴於介面則不管物件實際上是哪個類型（類別）的實例，只要知道物件實作了哪個介面就可以了，比抽象類別的依賴多了一些彈性。

8.3 接下來的主題

每一個章節的內容由淺至深，初學者該掌握的深度要到哪呢？在這個章節中，對於初學者我建議至少掌握以下幾點內容：

- 知道如何擴充（extends）類別
- 瞭解 "public"、"protected"、"private" 權限設定
- 知道如何重新定義方法
- 知道 Object 是 Java 中所有類別的父類別
- 瞭解 "final" 的用途
- 會定義抽象類別及實現抽象類別
- 會定義介面及實現介面

下一個章節的內容與類別的管理有關，您所定義的每一個類別，在編譯過後實際上都會以一個 .class 檔案儲存下來，該如何管理這些檔案，在 Java 中有一些機制可以使用，像是套件（Package）管理，您也可以看到一些不同的類別定義方式，像是「內部類別」（Inner class）、「匿名類別」（Anonymous class）等等。

第 9 章 管理類別檔案

在 Java 中每一個定義好的類別，在編譯過後都會以一個副檔名為 .class 的檔案儲存下來，在程式規模逐漸擴大之後，所需的類別將是以成千成萬的方式來計算，這麼多的類別檔案如果只是堆在一個目錄下加以管理，很容易發生名稱相同的衝突，要搜尋某個類別會是件麻煩事，管理這麼多散落一地似的類別檔案更是困難。

在這個章節中，您會認識到類別檔案的管理方式，像是各種內部類別所生成的類別檔案、使用「套件」（package）以階層方式來管理類別，您會認識到 "import" 的實質意義只是讓編譯器瞭解如何尋找目標類別，並且會學到 J2SE 5.0 中新增的靜態 import（static import）功能。

9.1 內部類別

在類別中您還可以再定義類別，稱之為「內部類別」（Inner class）或「巢狀類別」（Nested class）。非靜態的內部類別可以分為三種：「成員內部類別」（Member inner class）、「區域內部類別」（Local inner class）與「匿名內部類別」（Anonymous inner class）。內部類別的主要目的，都是對外部隱藏類別的存在性。

9.1.1 成員內部類別、區域內部類別

使用內部類別有幾個好處，其一是內部類別可以直接存取其所在類別中的私用（private）成員；其二是當某個 Slave 類別完全只服務於一個 Master 類別時，您可以將之設定為內部類別，如此使用 Master 類別的人就不用知道 Slave 的存在；再者是像在「靜態工廠」（Static factory）模式中，對呼叫靜態方法的物件隱藏返回物件的實作細節或產生方式。

先來介紹成員內部類別，基本上是在一個類別中直接宣告另一個類別，例如：

```
public class OuterClass {
    // 內部類別
    private class InnerClass {
        // ...
    }
}
```

成員內部類別同樣也可以使用 "public"、"protected" 或 "private" 來修飾其存取權限，範例 9.1 簡單示範成員內部類別的使用。

範例 9.1 PointDemo.java

```
public class PointDemo {
    // 內部類別
    private class Point {
        private int x, y;

        public Point() {}

        public void setPoint(int x, int y) {
            this.x = x;
            this.y = y;
        }

        public int getX() { return x; }
        public int getY() { return y; }
    }

    private Point[] points;

    public PointDemo(int length) {
        points = new Point[length];

        for(int i = 0; i < points.length; i++) {
            points[i] = new Point();
            points[i].setPoint(i*5, i*5);
        }
    }

    public void showPoints() {
        for(int i = 0; i < points.length; i++) {
            System.out.printf("Point[%d]: x = %d, y = %d\n",
                i, points[i].getX(), points[i].getY());
        }
    }
}
```

程式中假設 Point 類別只服務於 PointDemo 類別，外界不必知道 Point 類別的存在，只要知道如何操作 PointDemo 的實例就可以了，就像範例 9.2 所示。

範例 9.2 PointShow.java

```
public class PointShow {
    public static void main(String[] args) {
        PointDemo demo = new PointDemo(5);

        demo.showPoints();
    }
}
```

執行結果：

```
Point[0]: x = 0, y = 0
Point[1]: x = 5, y = 5
Point[2]: x = 10, y = 10
Point[3]: x = 15, y = 15
Point[4]: x = 20, y = 20
```

在檔案管理方面，成員內部類別在編譯完成之後，所產生的檔案名稱為「外部類別名稱\$內部類別名稱.class」，所以範例 9.1 在編譯過後會產生兩個檔案：PointDemo.class 與 PointDemo\$Point.class。

區域內部類別的使用與成員內部類別類似，區域內部類別定義於一個方法中，類別的可視範圍與生成之物件僅止於該方法之中。

內部類別還可以被宣告為 "static"，不過由於是 "static"，它不能存取外部類別的方法，而必須透過外部類別所生成的物件來進行呼叫，被宣告為 static 的內部類別，事實上也可以看作是另一種名稱空間的管理方式，例如：

```
public class Outer {
    public static class Inner {
        ...
    }
    ...
}
```

您可以如以下的方式來使用 Inner 類別：

```
Outer.Inner inner = new Outer.Inner();
```

良葛格的話匣子 有關內部類別的實際應用，建議看一下「靜態工廠模式」，對於使用靜態工廠類別的物件，它不需要知道返回物件的實例是如何產生的，只需要知道如何操作返回的物件，所以您可以宣告一個介面，並在靜態工廠中定義一個實作該介面的內部類別，由該內部類別產生實例，這是內部類別的一個應用場合。

另外也可以看一下「Iterator 模式」，一個實例是您在 Java SE 的 API 文件中找不到實作 java.util.Iterator 介面的類別，因為實作 Iterator 介面的類別是定義在集合類別（像是 java.util.ArrayList）之中，您只需要知道如何操作 Iterator 介面就可以了。

9.1.2 匿名內部類別

內部匿名類別可以不宣告類別名稱，而使用 "new" 直接產生一個物件，內部匿名類別可以是繼承某個類別或是實作某個介面，內部匿名類別的宣告方式如下：

```
new [類別或介面()] {
    // 實作
}
```

一個使用內部匿名類別的例子如範例7.3所示，您直接繼承 Object 類別定義一個匿名類別，重新定義 `toString()` 方法，並使用匿名類別直接產生一個物件。

範例 9.3 AnonymousClassDemo.java

```
public class AnonymousClassDemo {
    public static void main(String[] args) {
        Object obj =
            new Object() {
                public String toString() { // 重新定義toString()
                    return "匿名類別物件";
                }
            };
        System.out.println(obj);
    }
}
```

使用 `System.out.println()` 時如果傳入的是物件，會呼叫物件的 `toString()` 方法得到 `String` 實例，所以範例的輸出如下所示：

匿名類別物件

注意如果要在內部匿名類別中使用外部的區域變數，變數在宣告時必須為 "final"，例如下面的陳述是無法通過編譯的：

```
.....
public void someMethod() {
    int x = 10; // 沒有宣告final
    Object obj =
        new Object() {
            public String toString() {
                return String.valueOf(x); // x不可在匿名類別中使用
            }
        };
    System.out.println(obj);
}
....
```

在進行編譯時，編譯器會回報以下的錯誤：

```
local variable x is accessed from within inner class; needs to be declared final
```

您要在宣告區域變數x時加上"final"才可以通過編譯：

```
.....
public void someMethod() {
    final int x = 10; // 宣告final
    Object obj =
        new Object() {
            public String toString() {
                return String.valueOf(x); // x可在匿名類別中使用
            }
        };
    System.out.println(obj);
}
....
```

為什麼要加上 "final" 宣告呢？原因在於區域變數 x 並不是真正被拿來於內部匿名類別中使用，x 會被匿名類別複製作作為資料成員來使用，由於真正在匿名類別中的 x 是複製品，即使您在內部匿名類別中對 x 作了修改（例如 x=10 的指定），也不會影響真正的區域變數 x，事實上您也通不過編譯器的檢查，因為編譯器會要求您加上 "final" 關鍵字，這樣您就知道不能在內部匿名類別中改變 x 的值，因為即使能改變也沒有意義。

在檔案管理方面，內部匿名類別在編譯完成之後會產生「外部類別名稱\$編號.class」，編號為 1、2、3...n，每個編號 n 的檔案對應於第 n 個匿名類別，所以範例 9.3 編譯完成後，會產生 AnonymousClassDemo.class 與 AnonymousClassDemo\$1.class 兩個檔案。

9.2 package 與 import

隨著程式架構越來越大，類別個數越來越多，您會發現管理程式中維護類別名稱也會是一件麻煩的事，尤其是一些同名問題的發生，例如在程式中，您也許會定義一個 Point 類別，但另一個與您合作開發程式的開發人員並不曉得已經有這個類別名稱的存在，他可能也定義了一個 Point 類別，於是編譯過後他的 Point 類別檔案會覆蓋您的 Point 類別檔案，問題就這麼發生了。

9.2.1 設定套件 (package)

Java 提供「套件」 (package) 來管理類別，套件被設計與檔案系統結構相對應，如果您的套件設定為onlyfun.caterpillar，則該類別應該在 Classpath 可以存取到的路徑下的 onlyfun 目錄下之 caterpillar 目錄找到，沒有設定套件管理的類別會歸為「預設套件」 (default package) 。

為了要能建立與套件相對應的檔案系統結構，您在編譯時可以加入 "-d" 參數，並指定產生的類別檔案要儲存在哪一個目錄之下，實際使用範例來進行說明，在 Java 中定義套件時，要使用關鍵字 "package"，使用方法如範例 9.4。

範例 9.4 PackageDemo.java

```
package onlyfun.caterpillar;

public class PackageDemo {
    public static void main(String[] args) {
        System.out.println("Hello! World!");
    }
}
```

在編譯時您使用以下的指令指定編譯過後的類別檔案儲存目錄，'.'表示建立在目前的工作位置：

```
javac -d . UsePackage.java
```

編譯完成之後，在目前的工作位置中會出現 onlyfun 目錄，而 onlyfun 目錄下會有個 caterpillar 目錄，而 caterpillar 目錄下則有一個 PackageDemo.class 檔案，在編譯完成之後，"package" 的設定會成為類別名稱的一部份，也就是完整的類別名稱是 onlyfun.caterpillar.PackageDemo，所以在執行時要這麼下指令以指定類別名稱：

```
java onlyfun.caterpillar.PackageDemo
```

執行結果會出現"Hello! World!"。再來舉個例子，假設您如範例 9.5 建立了onlyfun.caterpillar.Point2D 類別。

範例 9.5 Point2D.java

```
package onlyfun.caterpillar;

public class Point2D {
    private int x;
    private int y;

    public Point2D() {
    }

    public Point2D(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

```

    public int getX() { return x; }
    public int getY() { return y; }
}

```

先用以下的指令來編譯 Point2D 類別：

```
javac -d . Point2D.java
```

在編譯過後，在 onlyfun 目錄的 caterpillar 目錄下會有 Point2D.class 檔案，而 "package" 所設定的名稱就成為類別名稱的一部份，也就是完整的類別名稱是 onlyfun.caterpillar.Point2D，除非您改變套件名稱並重新編譯類別，否則的話無法改變這個名稱，為了要使用 onlyfun.caterpillar.Point2D 類別，方法之一是使用「完全描述」（Fully qualified）名稱，也就是完整的指出「套件加類別」名稱，如範例 9.6 所示。

範例 9.6 Point2DDemo.java

```

public class Point2DDemo {
    public static void main(String[] args) {
        onlyfun.caterpillar.Point2D p1 = new
            onlyfun.caterpillar.Point2D(10, 20);

        System.out.printf("p1: (x, y) = (%d, %d)%n",
            p1.getX(), p1.getY());
    }
}

```

執行結果如下所示：

```
p1: (x, y) = (10, 20)
```

設定了套件名稱的類別，必須放置在對應的目錄中，例如若 Point2D 的套件設定為 onlyfun.caterpillar，則最後編譯完成的 .class 檔案必須放在 onlyfun 目錄的 caterpillar 目錄下，否則編譯時會有以下的錯誤發生：

```

bad class file: .\Point2D.class
class file contains wrong class: onlyfun.caterpillar.Point2D
Please remove or make sure it appears in the correct subdirectory of the classpath.
    Point2D p1 = new Point2D(10, 20);
    ^
1 error

```

如果您在編譯時沒有使用 "-d" 並指定 .class 檔案產生的目標目錄，則編譯完成之後，您也要自己建立對應於套件的目錄結構，然後將 .class 放入對應的目錄之下。

良葛格的話匣子 在命名套件時，可以使用組織的網域名稱來作為開頭命名，通常是倒過來命名，例如網域如果是 openhome.cc，則命名套件時可以用 cc.openhome，之後再加上您自己設定的套件名稱，這麼一來同名的衝突機會可以更少。

9.2.2 import 的意義

如果您有使用 "package" 來為您的類別設定套件管理，則編譯過後 "package" 所設定的名稱就成為類別名稱的一部份，在範例 9.6 中，您使用「完全描述」（Fully qualified）名稱來指定使用的類別，當然這個方法要打一長串的文字，因而使用上不是很方便，您可以使用 "import" 關鍵字，告知編譯器您所要使用的類別是位於哪一個套件，如此您可以少打一些字，讓編譯器多作一些事，例如範例 9.6 可以改寫為範例 9.7。

範例 9.7 Point2DDemo2.java

```
import onlyfun.caterpillar.Point2D;

public class Point2DDemo2 {
    public static void main(String[] args) {
        Point2D p1 = new Point2D(10, 20);

        System.out.printf("p1: (x, y) = (%d, %d)%n",
                           p1.getX(), p1.getY());
    }
}
```

雖然在範例 9.7 中新建 `onlyfun.caterpillar.Point2D` 物件時，只指定了 `Point2D` 名稱，但編譯器從一開頭的 "import" 設定得知，完整的類別名稱應該是 `onlyfun.caterpillar.Point2D`，因而可以順利編譯，執行結果與範例 9.6 是相同的。

在使用 "import" 時可以指定類別的完整描述，如果您會使用到某個套件下的許多類別，在使用 "import" 指定時，可以於套件指定後加上 '*'，這表示您會使用到該套件下的某些類別，編譯器會自己試著找出類別，例如範例 9.7 還可以再改為範例 9.8 的寫法。

範例 9.8 Point2DDemo3.java

```
import onlyfun.caterpillar.*;

public class Point2DDemo3 {
    public static void main(String[] args) {
        Point2D p1 = new Point2D(10, 20);

        System.out.printf("p1: (x, y) = (%d, %d)%n",
                           p1.getX(), p1.getY());
    }
}
```

編譯器在處理這個程式時，會先試著在現行工作路徑下找有無 `Point2D` 類別，如果找不到的話，編譯器會試著組合 "import" 上的設定來找尋 `Point2D` 類別，就範例 9.8 而言，會將 `onlyfun.caterpillar` 與 `Point2D` 組合在一起，然後試著找到 `onlyfun.caterpillar.Point2D` 類別。

您也許會發現無法編譯範例 9.8，可能出現以下的錯誤訊息：

```
bad class file: .\Point2D.java
file does not contain class Point2D
Please remove or make sure it appears in the correct subdirectory of the classpath.
```

這不是程式撰寫有誤，而是因為您 "import" 時使用了 '*'，並且您的 `Point2D.java` 原始檔案也在同一個目錄，照之前編譯器尋找類別順序的說明，編譯器會先找到 `Point2D.java`，但發現當中有設定套件，而 `Point2D.java` 沒有在對應的目錄 (`onlyfun/caterpillar`) 下，所以編譯器認定這是個錯誤。

將原始碼與編譯完成的類別檔放在一起容易發生這類的問題，事實上將原始碼與編譯完成的檔案放在一起並不是一個好的管理方式，您可以建一個專門放原始碼 `.java` 檔案的目錄 `src`，並建一個專門放 `.class` 檔案的目錄 `classes`，編譯時這麼下指令：

```
javac -d ./classes ./src/*.java
```

這麼一來產生的 `.class` 就會儲存在 `classes` 目錄下，可以至 `classes` 目錄下直接執行程式：

```
java Point2DDemo3
```

或者直接在執行編譯時的工作目錄下，以指定 Classpath 的方式如下執行程式：

```
java -cp ./classes Point2DDemo3
```

但要注意的是，如果您使用 "import" 之後，出現類別名稱有同名衝突時，編譯器就不知道如何處理了，例如：

```
import java.util.Arrays;
import onlyfun.caterpillar.Arrays;
public class SomeClass {
    ...
}
```

在這個例子中，編譯器從 "import" 上發現有兩個可能的 Arrays 類別，它不確定若遇到 Arrays 時您要使用的是 java.util.Arrays，或是 onlyfun.caterpillar.Arrays，編譯器只好回報錯誤訊息：

```
java.util.Arrays is already defined in a single-type import
import onlyfun.caterpillar.Arrays;
^
1 error
```

這個時候您就要考慮換一下類別名稱了（如果您有權更動那些類別的話），或者是不使用 "import"，直接使用完整描述；在 "import" 時儘量不因為貪圖方便而使用 '*'，也可以減少這種情況的發生。

良葛格的話匣子 使用 "import" 就是在告知編譯器您的類別位於哪一個套件下，而編譯器尋找類別最先是根據 Classpath 的設定，所以您也要瞭解 Classpath 的設定，建議您也看看官方網站上的 Classpath 設定文章，您對套件的瞭解會更深入：

- [Setting the class path \(Windows\)](#)
- [Setting the class path \(Solaris and Linux\)](#)

Java SE 平台的 .class 檔案是儲存在 JRE 安裝目錄的 /lib 下的 rt.jar 中，而額外的第三方（Third- party）元件可以放 /lib/ext 中，以及您自己設定的 Classpath。

9.2.3 public 與套件

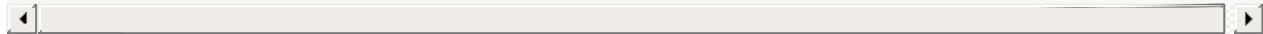
一個類別在定義時可以使用 "public" 加以修飾，一個 .java 檔案中可以定義數個類別，但只能有一個被宣告為 "public"，沒有被宣告為 "public" 的類別只能被同一個套件中的類別之實例呼叫使用，例如將範例 9.5 中 onlyfun.caterpillar.Point2D 類別上的 "public" 拿掉並重新編譯，接著再編譯 Point2DDemo.java 檔案時，會出現以下的錯誤，因為 Point2DDemo（預設套件）與 onlyfun.caterpillar.Point2D 不在同一個套件：

```
Point2DDemo.java:3: onlyfun.caterpillar.Point2D is not public in onlyfun.caterpillar;      cannot be accessed from outside its enclosing type
        onlyfun.caterpillar.Point2D p1 = new
```

類別成員也可以宣告為 "public"，宣告為 "public" 的類別成員可以被其它物件呼叫使用，如果宣告類別時不使用 "public"、"protected" 或 "private" 設定權限，則預設為「套件存取範圍」，只有同一個套件中的類別可以呼叫這些類別成員，例如範例 9.5 中將 getX()、getY() 上的 "public" 拿掉並重新編譯，接著再編譯 Point2DDemo.java 時，會出現以下的錯誤，因為 Point2DDemo（預設套件）與 onlyfun.caterpillar.Point2D 不在同一個套件：

```
Point2DDemo.java:7: getX() is not public in onlyfun.caterpillar.Point2D; cannot be accessed from outside package
    p1.getX(), p1.getY());
           ^

```



類別上的權限設定會約束類別成員上的權限設定，所以如果類別上不宣告 "public"，而類別成員上設定了 "public"，則類別成員同樣的也只能被同一個套件的類別存取，也就是說如果您這麼撰寫程式：

```
package onlyfun.caterpillar;
class SomeClass {
    // ...
    public void someMethod() {
        // ...
    }
}
```

其效果等同於：

```
package onlyfun.caterpillar;
class SomeClass {
    // ...
    void someMethod() {
        // ...
    }
}
```

由這邊的討論，可以再來看看預設建構方法的權限。首先要知道的是，當您在 Java 中定義一個類別，但沒有定義建構方法時，編譯器會自動幫您產生一個預設建構方法，也就是說，如果您這麼寫：

```
package onlyfun.caterpillar;
public class Test {
    ...
}
```

則編譯器會自動加上預設建構方法，也就是相當於這麼寫：

```
package onlyfun.caterpillar;
public class Test {
    public Test() {
    }
    ...
}
```

如果您自行定義建構方法，則編譯器就不會幫您加上預設建構方法，所以當您這麼定義時：

```
package onlyfun.caterpillar;
public class Test {
    public Test(int i) {
        ...
    }
    ...
}
```

則在建構時，就必須指明使用哪個建構方法，簡單的說，您就不能使用以下的方式來建構：

```
Test test = new Test();
```

有時會建議即使沒有用到，在定義自己的建構方法的同時，也加上個沒有參數的建構方法，例如：

```
package onlyfun.caterpillar;
public class Test {
    public Test() { // 即使沒用到，也先建立一個空的建構方法
    }

    public Test(int i) {
        ...
    }
    ...
}
```

要注意的是，在繼承時，如果您沒有使用 super() 指定要使用父類別的哪個建構方法，則預設會尋找父類別中無參數的建構方法。

預設建構方法的存取權限是跟隨著類別的存取權限而設定，例如：

```
package onlyfun.caterpillar;
public class Test {
}
```

由於類別宣告為 public，所以預設建構方法存取權限為 public。如果是以下的話：

```
package onlyfun.caterpillar;
class Test {
}
```

則預設建構方法存取權限為套件存取權限，也就是編譯器會自動為您擴展為：

```
package onlyfun.caterpillar;
class Test {
    Test() {
    }
}
```

在這邊整理一下 private、protected、public 與 default 與類別及套件的存取關係：

表 9.1 權限設定為套件的關係

存取修飾	同一類別	同一套件	子類別	全域
private	OK			
(default)	OK	OK		
protected	OK	OK	OK	
public	OK	OK	OK	OK

9.2.4 import 靜態成員

在 J2SE 5.0 後新增了 "import static" 語法，它的作用與 "import" 類似，都是為了讓您可以省一些打字功夫，讓編譯器多作一

點事而存在的。“import static” 是使用時的語法，原文的文章或原文書中介紹這個功能時，大都用 “static import” 描述這個功能，編譯器訊息也這麼寫，但為了比較彰顯這個功能的作用，這邊稱之為「import 靜態成員」。

使用 “import static” 語法可以讓您 “import” 類別或介面中的靜態成員，一個實際的例子如範例 9.9 所示。

範例 9.9 HelloWorld.java

```
import static java.lang.System.out;

public class HelloWorld {
    public static void main(String[] args) {
        out.println("Hello! World!");
    }
}
```

在範例中您將 `java.lang.System` 類別中的 `out` 靜態成員 “import” 至程式中，編譯時編譯器遇到 `out` 名稱，可以從 “import static” 上知道 `out` 是 `System` 中的靜態成員，因而自動展開為 `System.out` 並加以編譯。

再來看一個例子，`Arrays` 類別中有很多的靜態方法，為了使用方便，您可以使用 “import static” 將這些靜態方法 “import” 至程式中，如範例 9.10 所示。

範例 9.10 ImportStaticDemo.java

```
import static java.lang.System.out;
import static java.util.Arrays.sort;

public class ImportStaticDemo {
    public static void main(String[] args) {
        int[] array = {2, 5, 3, 1, 7, 6, 8};

        sort(array);

        for(int i : array) {
            out.print(i + " ");
        }
    }
}
```

編譯器遇到範例 9.10 中的 `sort()` 方法，可以從 “import static” 上知道 `sort()` 方法是 `java.util.Arrays` 上的靜態成員，執行結果如下：

```
1 2 3 5 6 7 8
```

如果您想要 “import” 類別下所有的靜態成員，也可以使用 “*” 字元，例如將範例 9.10 中的 “import static” 改為以下也是可行的：

```
`import static java.util.Arrays.*;
```

“import static” 語法可以讓您少打一些字，但是您要注意名稱衝突問題，對於名稱衝突編譯器可能透過以下的幾個方法來解決：

- 成員覆蓋

如果類別中有同名的資料成員或方法，則優先選用它們。

- 區域變數覆蓋

如果方法中有同名的變數名或參數名，則選用它們。

- 重載（Overload）方法上的比對

嘗試使用重載機制判斷，也就是透過方法名稱及參數列的比對來選擇適當的方法。

如果編譯器無法判斷，則會回報錯誤，例如若您定義了如下的類別：

```
package onlyfun.caterpillar;
public class Arrays {
    public static void sort(int[] arr) {
        // ...
    }
}
```

然後如下撰寫程式：

```
import static java.lang.System.out;
import static java.util.Arrays.sort;
import static onlyfun.caterpillar.Arrays.sort;
public class ImportStaticDemo2 {
    public static void main(String[] args) {
        int[] array = {2, 5, 3, 1, 7, 6, 8};
        sort(array);
        for(int i : array) {
            out.print(i + " ");
        }
    }
}
```

由於從 `java.util.Arrays.sort` 與 `onlyfun.caterpillar.Arrays.sort` 的兩行 "import static" 上都可以找到 `sort`，編譯器無法辦別要使用哪一個 `sort()` 方法，因而編譯時會出現以下的錯誤：

```
ImportStaticDemo2.java:9: reference to sort is ambiguous, both method sort(int[]) in      onlyfun.caterpillar.Arrays and
      sort(array);
           ^
1 error
```

9.3 接下來的主題

每一個章節的內容由淺至深，初學者該掌握的深度要到哪呢？在這個章節中，對於初學者我建議至少掌握以下幾點內容：

- 會定義成員內部類別
- 會使用匿名內部類別
- 知道內部類別編譯過後的 .class 檔名稱命名方式
- 會使用套件整理類別並瞭解其與實體目錄之關係
- 知道"public"與套件的權限關係
- 瞭解"import"、"import static" 的真正目的

一個程式的撰寫的過程中，如何避免程式執行時的錯誤，往往佔了程式開發時程的絕大多數時間，對於避免程式執行時的錯誤，Java 提供了例外處理機制，事實上到目前為止，您可能已經遇過不少 Exception 的實例了，下一個章節將說明這些例外是什麼意義，以及如何處理這些實例。

第 10 章 例外處理 (Exception Handling)

程式中的臭蟲（Bug）總是無所不在，即使您認為程式中應該沒有錯誤了，臭蟲總會在某個時候鑽出來困擾您，面對程式中各種層出不窮的錯誤，Java 提供了「例外處理」機制來協助開發人員避開可能的錯誤，「例外」（Exception）在 Java 中代表一個錯誤的實例，編譯器會幫您檢查一些可能產生例外（Checked exception）的狀況，並要求您注意並處理，而對於「執行時期例外」（Runtime exception），您也可以嘗試捕捉例外並將程式回復至正常狀況。

這個章節會介紹 Java 的例外處理架構以及斷言（Assertion），雖然就章節內容而言，這章是個相對簡短的章節，但例外處理卻可能是您撰寫程式時最常面對的議題，因為程式中的錯誤無所不在，編譯器會經常提醒您必須作例外處理，而您也經常必須處理執行時期例外。

10.1 例外處理入門

回憶一下 6.2.1 中介紹過的「命令列引數」（Command line argument），在還沒有學會使用 Java 的例外處理之前，為了確定使用者是否有提供引數，您可能要先檢查引數的陣列長度是否為 0，以免程式存取超過陣列長度而發生錯誤：

```
if(args.length == 0) {
    // 使用者沒有指定引數，顯示引數功能畫面
}
else {
    // 執行所指定引數的對應功能
}
```

利用條件判斷式來避免錯誤的發生，這樣的檢查方式在一些程式語言中經常出現，然而顯然的，處理錯誤的邏輯與處理業務的邏輯混在一起，如果更多的錯誤狀況必須檢查的話，程式會更難以閱讀，且由於使用了一些判斷式，即使發生機率低的錯誤，也都必須一視同仁的進行判斷檢查，這會使得程式的執行效能受到一定程度的影響。

Java 的例外處理機制可以協助您避開或是處理程式可能發生的錯誤，「例外」（Exception）在 Java 中代表一個錯誤的實體物件，在特定錯誤發生時會丟出特定的例外物件，有些預期中可能發生的例外，編譯器會提醒您先行處理，對於一些程式運行時所發生的執行時期例外，您有機會捕捉這些例外，並嘗試將程式回復至正常運作狀態。

在 Java 中如果想嘗試捕捉例外，可以使用 "try"、"catch"、"finally" 三個關鍵字組合的語法來達到，其語法基本結構如下：

```
try {
    // 陳述句
}
catch(例外型態 名稱) {
    // 例外處理
}
finally {
    // 一定會處理的區塊
}
```

一個 "try" 語法所包括的區塊，必須有對應的 "catch" 區塊或是 "finally" 區塊，"try" 區塊可以搭配多個 "catch" 區塊，如果有設定 "catch" 區塊，則 "finally" 區塊可有可無，如果沒有定義 "catch" 區塊，則一定要有 "finally" 區塊。

使用實例來說明，您可以使用 try...catch 語法來取代命令列引數的陣列長度檢查動作，如範例 10.1 所示。

範例 10.1 CheckArgsDemo.java

```
public class CheckArgsDemo {
    public static void main(String[] args) {
        try {
            System.out.printf("執行 %s 功能%n", args[0]);
        }
        catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("沒有指定引數");
            e.printStackTrace();
        }
    }
}
```

如果在執行程式時沒有指定引數，那麼 args 陣列的長度是 0，程式中嘗試從 args[0] 取得引數時就會發生錯誤，錯誤的實例是 ArrayIndexOutOfBoundsException，這個實例會在被對應的 "catch" 所捕捉，在範例 10.1 中被捕獲的例外指定給 e 名稱來參考，例外被捕獲後會執行對應的 "catch" 區塊，在範例中是顯示提示訊息，並使用 printStackTrace() 會顯示完整的例外訊息，沒有指定引數時的執行結果如下：

```
>java CheckArgsDemo  
沒有指定引數  
java.lang.ArrayIndexOutOfBoundsException: 0  
    at CheckArgsDemo.main(CheckArgsDemo.java:4)
```

範例 10.1 中並沒有使用條件判斷式來檢查陣列長度，也就是沒有使用 if 陳述句，例外處理只有在錯誤真正發生，也就是丟出例外時才處理，所以與使用 if 判斷式每次都要進行檢查動作相比，效率上會好一些，要注意的是，例外處理最好只用於錯誤處理，而不應是用於程式業務邏輯的一部份，因為例外的產生要消耗資源，例如以下應用例外處理的方式就不適當：

```
while(true) {  
    try {  
        System.out.println(args[i]);  
        i++;  
    }  
    catch(ArrayIndexOutOfBoundsException e) {  
        break;  
    }  
}
```

循序取出陣列值時，最後一定會到達陣列的邊界，檢查邊界是必要的動作，是程式業務邏輯的一部份，而不是錯誤處理邏輯的一部份，您該使用的是 for 迴圈而不是依賴例外處理，例如下面的方式才是正確的：

```
for(int i = 0; i < args.length; i++) {  
    System.out.println(args[i]);  
}
```

10.2 受檢例外（Checked Exception）、執行時期例外（Runtime Exception）

在某些情況下例外的發生是可預期的，例如使用輸入輸出功能時，可能會由於硬體環境問題，而使得程式無法正常從硬體取得輸入或進行輸出，這種錯誤是可預期發生的，像這類的例外稱之為「受檢例外」（Checked Exception），對於受檢例外編譯器會要求您進行例外處理，例如在使用 `java.io.BufferedReader` 的 `readLine()` 方法取得使用者輸入時，編譯器會要求您於程式碼中明確告知如何處理 `java.io.IOException`，範例 10.2 是個簡單的示範。

範例 10.2 CheckedExceptionDemo.java

```
import java.io.*;

public class CheckedExceptionDemo {
    public static void main(String[] args) {
        try {
            BufferedReader buf = new BufferedReader(
                new InputStreamReader(System.in));
            System.out.print("請輸入整數: ");
            int input = Integer.parseInt(buf.readLine());
            System.out.println("input x 10 = " + (input*10));
        }
        catch(IOException e) { // checked exception
            System.out.println("I/O錯誤");
        }
        catch(NumberFormatException e) { // runtime exception
            System.out.println("輸入必須為整數");
        }
    }
}
```

`IOException` 是受檢例外，是可預期會發生的例外，編譯器要求您必須處理，如果您不在程式中處理的話，例如將 `IOException` 的 "catch" 區塊拿掉，編譯器會回報錯誤訊息：

```
CheckedExceptionDemo.java:9: unreported exception java.io.IOException; must be caught or declared to be thrown
```

範例 10.2 中試著從使用者輸入取得一個整數值，由 `BufferedReader` 物件所讀取到的輸入是個字串，您使用 `Integer` 類別的 `parseInt()` 方法試著剖析該字串為整數，如果無法剖析，則會發生錯誤並丟出一個 `NumberFormatException` 例外物件，當這個例外丟出後，程式會離開目前執行的位置，而如果設定的 "catch" 有捕捉這個例外，則會執行對應區塊中的陳述句，注意當例外一但丟出，就不會再回到例外的丟出點了。

像 `NumberFormatException` 例外是「執行時期例外」（Runtime exception），也就是例外是發生在程式執行期間，並不一定可預期它的發生，編譯器不要求您一定要處理，對於執行時期例外若沒有處理，則例外會一直往外丟，最後由 JVM 來處理例外，JVM 所作的就是顯示例外堆疊訊息，之後結束程式。

如果 `try...catch` 後設定有 "finally" 區塊，則無論例外是否有發生，都一定會執行 "finally" 區塊。

10.3 throw、throws

當程式發生錯誤而無法處理的時候，會丟出對應的例外物件，除此之外，在某些時刻，您可能會想要自行丟出例外，例如在捕捉例外並處理結束後，再將例外丟出，讓下一層例外處理區塊來捕捉；另一個狀況是重新包裝例外，將捕捉到的例外以您自己定義的例外物件加以包裝丟出。若想要自行丟出例外，您可以使用 "throw" 關鍵字，並生成指定的例外物件，例如：

```
throw new ArithmeticException();
```

舉個例子來說明，在 Java 的除法中，允許浮點數運算時除數為 0，所得到的結果是 Infinity，也就是無窮數，如果您想要自行檢驗除零錯誤，可以自行丟出 ArithmeticException 例外，這個例外是在整數除法且為除數 0 時所引發的例外，您可以讓浮點數運算除數為 0 時也丟出這個例外。

範例 10.3 ThrowDemo.java

```
public class ThrowDemo {
    public static void main(String[] args) {
        try {
            double data = 100 / 0.0;
            System.out.println("浮點數除以零：" + data);
            if(String.valueOf(data).equals("Infinity"))
                throw new ArithmeticException("除零例外");
        }
        catch(ArithmeticException e) {
            System.out.println(e);
        }
    }
}
```

在檢驗運算結果為 Infinity 時，您自行建立 ArithmeticException 實例並使用 "throw" 丟出，產生實例的同時您可以指定訊息，執行結果如下：

```
浮點數除以零：Infinity
java.lang.ArithmetricException: 除零例外
```

在巢狀的 try...catch 結構時，必須注意該例外是由何者引發並由何者捕捉，例如：

範例 10.4 CatchWho.java

```
public class CatchWho {
    public static void main(String[] args) {
        try {
            try {
                throw new ArrayIndexOutOfBoundsException();
            }
            catch(ArrayIndexOutOfBoundsException e) {
                System.out.println(
                    "ArrayIndexOutOfBoundsException" +
                    "/內層try-catch");
            }

            throw new ArithmeticException();
        }
        catch(ArithmetricException e) {
            System.out.println("發生ArithmetricException");
        }
        catch(ArrayIndexOutOfBoundsException e) {
            System.out.println(
                "ArrayIndexOutOfBoundsException" +
```

```

        "/外層try-catch");
    }
}
}

```

執行結果：

```

ArrayIndexOutOfBoundsException/內層try-catch
發生ArithmeticException

```

在範例 10.4 中，丟出的 `ArrayIndexOutOfBoundsException` 由內層的 "catch" 先捕捉到，由於內層已經捕捉了例外，所以外層的 "catch" 並不會捕捉到 `ArrayIndexOutOfBoundsException` 例外，如果內層的 "catch" 並沒有捕捉到這個例外，則外層的 "catch" 就有機會捕捉這個例外，例如範例 10.5 中 `ArrayIndexOutOfBoundsException` 就會被外層的 "catch" 捕捉到。

範例 10.5 CatchWho2.java

```

public class CatchWho2 {
    public static void main(String[] args) {
        try {
            try {
                throw new ArrayIndexOutOfBoundsException();
            }
            catch(ArithmeticException e) {
                System.out.println(
                    "ArrayIndexOutOfBoundsException" +
                    "/內層try-catch");
            }

            throw new ArithmeticException();
        }
        catch(ArithmeticException e) {
            System.out.println("發生ArithmeticException");
        }
        catch(ArrayIndexOutOfBoundsException e) {
            System.out.println(
                "ArrayIndexOutOfBoundsException" +
                "/外層try-catch");
        }
    }
}

```

執行結果：

```

ArrayIndexOutOfBoundsException/外層try-catch

```

如果您在方法中會有例外的發生，而您並不想在方法中直接處理，而想要由呼叫方法的呼叫者來處理，則您可以使用 "throws" 關鍵字來宣告這個方法將會丟出例外，例如 `java.io.BufferedReader` 的 `readLine()` 方法就聲明會丟出 `java.io.IOException`。使用 "throws" 聲明丟出例外的時機，通常是工具類別的某個工具方法，因為作為被呼叫的工具，本身並不需要將處理例外的方式給定義下來，所以在方法上使用"throws"聲明會丟出例外，由呼叫者自行決定如何處理例外是比較合適的，您可以如下使用 "throws" 來丟出例外：

```

private void someMethod(int[] arr) throws
    ArrayIndexOutOfBoundsException,
    ArithmeticException {
    // 實作
}

```

注意方法上若會丟出多種可能的例外時，中間是使用逗點分隔；當方法上使用 "throws" 宣告丟出例外時，意味著呼叫該方法

的呼叫者必須處理這些例外，範例 10.6 是 "throws" 的簡單示範。

範例 10.6 ThrowsDemo.java

```
public class ThrowsDemo {  
    public static void main(String[] args) {  
        try {  
            throwsTest();  
        }  
        catch(ArithmetricException e) {  
            System.out.println("捕捉例外");  
        }  
    }  
  
    private static void throwsTest()  
        throws ArithmetricException {  
        System.out.println("這只是一個測試");  
        // 程式處理過程假設發生例外  
        throw new ArithmetricException();  
    }  
}
```

執行結果：

```
這只是一個測試  
捕捉例外
```

簡單的說，您要不就在方法中直接處理例外，要不就在方法上宣告該方法會丟回例外，由呼叫它的呼叫者來處理例外。

良葛格的話匣子 您也可以在定義介面（interface）時於方法上聲明"throws"某些類型的例外，然而要小心使用，因為若您在這些方法中發生了某些不是方法聲明的例外，您就無法將之"throw"，只能自行撰寫一些try..catch來暗自處理掉，或者是重新包裝例外為"throws"上所聲明的例外，或者是將該例外包裝為RuntimeException然後再丟出。

10.4 例外的繼承架構

Java 所處理的例外主要可分為兩大類：一種是嚴重的錯誤，例如硬體錯誤或記憶體不足等問題，與此相關的類別是位於 `java.lang` 下的 `Error` 類別及其子類別，對於這類的錯誤通常程式是無力自行回復；另一種是非嚴重的錯誤，代表可以處理的狀況，例如使用者輸入了不合格式的資料，這種錯誤程式有機會回復至正常運作狀況，與這類錯誤相關的類別是位於 `java.lang` 下的 `Exception` 類別及其子類別。

`Error` 類別與 `Exception` 類別都繼承自 `Throwable` 類別，`Throwable` 類別擁有幾個取得相關例外訊息的方法。

- `getLocalizedMessage()`

取得例外物件的區域化訊息描述

- `getMessage()`

取得例外物件的訊息描述

- `printStackTrace()`

顯示例外的堆疊訊息，這個方法在追蹤例外發生的根源時相當的有用，簡單的說若 A 方法中呼叫了 B 方法，而 B 方法中呼叫了 C 方法，C 方法產生了例外，則在處理這個例外時呼叫 `printStackTrace()` 可以得知整個方法呼叫的過程，由此得知例外是如何被層層丟出的。

除了使用這些方法之外，您也可以簡單的利用例外物件 `toString()` 方法取得例外的簡單訊息描述。

您所接觸的例外通常都是衍生自 `Exception` 類別，其中是有些「受檢例外」（Checked exception），例如 `ClassNotFoundException`（嘗試載入類別時失敗所引發，例如類別檔案不存在）、`InterruptedException`（執行緒非執行中而嘗試中斷所引發的例外）等，而有些是「執行時期例外」（Runtime exception），也稱「非受檢例外」（Uncckecked exception），例如 `ArithmetricException`、`ArrayIndexOutOfBoundsException` 等。以下列出一些重要的例外繼承架構：

```

Throwable
  Error (嚴重的系統錯誤)
    LinkageError
    ThreadDeath
    VirtualMachineError
    ...
Exception
  ClassNotFoundException
  CloneNotSupportedException
  IllegalAccessError
  ...
  RuntimeException (執行時期例外)
    ArithmetricException
    ArrayStoreException
    ClassCastException
  ...

```

`Exception` 下非 `RuntimeException` 衍生之例外類別如果有引發的可能，則您一定要在程式中明確的指定處理才可以通過編譯，因為這些例外是可預期的，編譯器會要求您明確處理，所以才稱之為「受檢例外」（Checked exception），例如當您使用到 `BufferedReader` 的 `readLine()` 時，由於有可能引發 `IOException` 這個受檢例外，您要不就在 `try...catch` 中處理，要不就在方法上使用 "throws" 表示由呼叫它的呼叫者來處理。

屬於 `RuntimeException` 衍生出來的類別是「執行時期例外」（Runtime exception），是在執行時期會發生的例外，這個例外不預期它一定會發生，端看程式邏輯寫的如何，因而不需要特別使用 `try...catch` 或是在方法上使用 "throws" 告宣也可以通過編譯，所以才稱之為「非受檢例外」（Unckecked exception），例如您在使用陣列時，並不一定要處理 `ArrayIndexOutOfBoundsException` 例外，因為只要程式邏輯寫的正確，這個例外就不會發生。

瞭解例外處理的繼承架構是必要的，例如在捕捉例外物件時，如果父類別例外物件在子類別例外物件之前被捕捉，則 "catch" 子類別例外物件的區塊將永遠不會被執行，事實上編譯器也會幫您檢查這個錯誤，例如：

範例 10.7 ExceptionDemo.java

```
import java.io.*;

public class ExceptionDemo {
    public static void main(String[] args) {
        try {
            throw new ArithmeticException("例外測試");
        }
        catch(Exception e) {
            System.out.println(e.toString());
        }
        catch(ArithmeticException e) {
            System.out.println(e.toString());
        }
    }
}
```

因為 `Exception` 是 `ArithmeticException` 的父類別，所以 `ArithmeticException` 的實例會先被 `Exception` 的 "catch" 區塊捕捉到，範例 10.7 在編譯時將會產生以下的錯誤訊息：

```
ExceptionDemo.java:11: exception java.lang.ArithmeticException has already been caught
    catch(ArithmeticException e) {
    ^
1 error
```

要完成這個程式的編譯，您必須更改例外物件捕捉的順序，如範例 10.8 所示。

範例 10.8 ExceptionDemo2.java

```
import java.io.*;

public class ExceptionDemo2 {
    public static void main(String[] args) {
        try {
            throw new ArithmeticException("例外測試");
        }
        catch(ArithmeticException e) {
            System.out.println(e.toString());
        }
        catch(Exception e) {
            System.out.println(e.toString());
        }
    }
}
```

在撰寫程式時，您也可以如範例 10.8 將 `Exception` 例外物件的捕捉撰寫在最後，以便捕捉到所有您所尚未考慮到的例外，在除蟲（Debug）階段時是很有用的。

如果您要自訂自己的例外類別，您可以繼承 `Exception` 類別而不是 `Error` 類別，`Error` 是屬於嚴重的系統錯誤，程式通常無力從這類的錯誤中回復，所以您不用去處理它，事實上在 Java 程式中也不希望您處理 `Error` 類別的例外。

如果使用繼承時，父類別的某個方法上宣告了 `throws` 某些例外，而在子類別中重新定義該方法時，您可以：

- 不處理例外（重新定義時不設定 `throws`）
- 可僅 `throws` 父類別中被重新定義的方法上之某些例外
- 可 `throws` 被重新定義的方法上之例外之子類別

但是您不可以：

- throws 父類別方法中未定義的其它例外
- throws 被重新定義的方法上之例外之父類別

10.5 斷言 (Assertion)

例外是程式中非預期的錯誤，例外處理是在這些錯誤發生時所採取的措施。有些時候，您預期程式中應該會處於何種狀態，例如某些情況下某個值必然是多少，這稱之為一種斷言。斷言有兩種結果：成立或不成立。當預期結果與實際執行相同時，斷言成立，否則斷言不成立。

Java 在 JDK 1.4 之後提供斷言陳述，有兩種使用的語法：

```
assert boolean_expression;
assert boolean_expression : detail_expression;
```

boolean_expression 如果為 true，則什麼事都不會發生，如果為 false，則會發生 `java.lang.AssertionError`，此時若採取的是第二個語法，則會將 detail_expression 的結果顯示出來，如果當中是個物件，則呼叫它的 `toString()` 顯示文字描述結果。

一個使用斷言的時機是內部不變量 (Internal invariant) 的判斷，例如在某個時間點上，或某個狀況發生時，您判斷某個變數必然要是某個值，舉個例子來說：

範例 10.9 AssertionDemo.java

```
public class AssertionDemo {
    public static void main(String[] args) {
        if(args.length > 0) {
            System.out.println(args[0]);
        }
        else {
            assert args.length == 0;
            System.out.println("沒有輸入引數");
        }
    }
}
```

在正常的預期中，陣列長度是不會小於 0 的，所以一但執行至 else 區塊，陣列長度必然只有一個可能，就是等於 0，您斷言 `args.length==0` 結果必然成立，else 之中的程式碼也只有在斷言成立的狀況下才能執行，如果不成立，表示程式運行存在錯誤，else 區塊不應被執行，您要停下來檢查程式的錯誤，事實上斷言主要的目的通常是在開發時期測試使用。

斷言功能是在 JDK 1.4 之後提供的，由於斷言使用 `assert` 作為關鍵字，為了避免您以前在 JDK 1.3 或更早之前版本的程式使用了 `assert` 作為變數，而導致的名稱衝突問題，預設上執行時是不啟動斷言檢查的，如果您要在執行時啟動斷言檢查，可以使用 `-enableassertions` 或是 `-ea` 引數，例如：

```
java -ea AssertionDemo
```

另一個使用斷言的時機為控制流程不變量 (Control flow invariant) 的判斷，例如在使用 `switch` 時：

```
switch(var) {
    case Constants.Con1:
        ...
        break;
    case Constants.Con2:
        ...
        break;
    case Constants.Con3:
        ...
        break;
    default:
        assert false : "非定義的常數";
```

{}

假設您已經在 switch 中列出了所有的常數，即 var 不該出現 Constants.Con1、Constants.Con2、 Constants.Con3 以外的常數，則如果發生 default 被執行的情況，表示程式的狀態與預期不符，此時由於 assert false，所以必然斷言失敗。

簡單的說，斷言是判定程式中的某個執行點必然是某個狀態，所以它不能當作像 if 之類的判斷式來使用，Assertion 不應被當做程式執行流程的一部份。

10.6 接下來的主題

每一個章節的內容由淺至深，初學者該掌握的深度要到哪呢？在這個章節中，對於初學者我建議至少掌握以下幾點內容：

- 會使用 try...catch...finally 語法
- 會使用 throw、throws 語法
- 瞭解受檢例外（Checked exception）與非受檢例外（Unchecked exception）的差別
- 知道例外的繼承架構與 try...catch 的捕捉順序

下一個章節要來看看 J2SE 5.0 新增的功能：列舉型態（Enumerated Types）。列舉型態可以解決常數設定的問題，並提供更多編譯時期的檢查，比 J2SE 1.4 或更早版本只使用變數宣告常數更為實用。

第 11 章 列舉型態 (Enumerated Types)

程式中經常會使用到一些常數，如果有些常數是共用的，在 Java 中可以定義一個類別或介面來統一管理常數，而其它物件從這些類別或介面上取用常數，如果需要修改常數則可以從這些類別或介面上直接修改，而不用更動到程式的其它部份，這種使用常數的方式在 J2SE 1.4 或之前的版本相當常見。

J2SE 5.0 中新增了「列舉型態」 (Enumerated Types)，您可以使用這個功能取代之前 J2SE 1.4 或之前版本定義常數的方式，除了常數設置的功能之外，列舉型態還給了您許多編譯時期的檢查功能，但別想的太複雜，列舉型態本質上還是以類別的方式存在，因而它提供了這麼多額外的功能並不奇怪。

11.1 常數設置與列舉型態

在瞭解 J2SE 5.0 中新增的「列舉型態」（Enumerated Types）功能之前，您可以先瞭解一下在 J2SE 1.4 或之前版本中，是如何定義共用常數的，如此在接觸列舉型態時，您就可以感覺它所帶來的更多好處。

11.1.1 常數設置

有時候您會需要定義一些常數供程式使用，您可以使用介面或類別來定義，例如您可以使用介面來定義操作時所需的共用常數，範例 11.1 是個簡單示範。

範例 11.1 ActionConstants.java

```
public interface ActionConstants {
    public static final int TURN_LEFT = 1;
    public static final int TURN_RIGHT = 2;
    public static final int SHOT = 3;
}
```

共用的常數通常是可以直接取用並且不可被修改的，所以您在宣告時加上 "static" 與 "final"，如此您可以在程式中直接使用像是 ActionConstants.TURN_LEFT 的名稱來取用常數值，例如：

```
public void someMethod() {
    ...
    doAction(ActionConstants.TURN_RIGHT);
    ...
}
public void doAction(int action) {
    switch(action) {
        case ActionConstants.TURN_LEFT:
            System.out.println("向左轉");
            break;
        case ActionConstants.TURN_RIGHT:
            System.out.println("向右轉");
            break;
        case ActionConstants.SHOT:
            System.out.println("射擊");
            break;
    }
}
```

如果使用類別來宣告的話，方法也是類似，例如：

範例 11.2 CommandTool.java

```
public class CommandTool {
    public static final String ADMIN = "onlyfun.caterpillar.admin";
    public static final String DEVELOPER = "onlyfun.caterpillar.developer";

    public void someMethod() {
        // ...
    }
}
```

如果常數只是在類別內部使用的話，就宣告其為 "private" 或是 "protected" 就可以了，宣告為類別外可取用的常數，通常是與類別功能相依的常數，例如使用 CommandTool 時若會使用到與 CommandTool 功能相依的常數的話，將這些常數直接宣告在 CommandTool 類別上取用時就很方便，而使用介面所宣告的常數，則通常是整個程式或某個模組中都會共用到的常數。

對於簡單的常數設置，上面的作法已經足夠了，在 J2SE 5.0 中則新增了「列舉型態」（Enumerated Types），使用列舉型態，除了簡單的常數設定功能之外，您還可以獲得像編譯時期型態檢查等更多的好處。

良葛格的話匣子 宣告常數時，通常使用大寫字母，並可以底線來區隔每個單字以利識別，例如像TURN_LEFT這樣的名稱。

11.1.2 列舉型態入門

您已經知道可以在類別或介面中宣告常數來統一管理常數，這只是讓您存取與管理常數方便而已，來看看這個例子：

```
public void someMethod() {
    ...
    doAction(ActionConstants.TURN_RIGHT);
    ...
}
public void doAction(int action) {
    switch(action) {
        case ActionConstants.TURN_LEFT:
            System.out.println("向左轉");
            break;
        ...
    }
}
```

這種作法本身沒錯，只不過 doAction() 方法接受的是int型態的常數，您沒有能力阻止程式設計人員對它輸入 ActionConstants 規定外的其它常數，也沒有檢查 "switch" 中列舉的值是不是正確的值，因為參數 action 就只是 int 型態而已，當然您可以自行設計一些檢查動作，這需要一些額外的工作，如果您使用 J2SE 5.0 中新增的「列舉型態」（Enumerated Types），就可以無需花額外的功夫就輕易的解決這些問題。

在 J2SE 5.0 中要定義列舉型態是使用 "enum" 關鍵字，以下先來看看列舉型態的應用，舉個實際的例子，範例 11.3 是定義了 Action 列舉型態。

範例 11.3 Action.java

```
public enum Action {
    TURN_LEFT,
    TURN_RIGHT,
    SHOOT
}
```

不用懷疑，在 Action.java 中撰寫範例 11.3 的內容然後編譯它，雖然語法上不像是在定義類別，但列舉型態骨子裏就是一個類別，所以您編譯完成後，會產生一個 Action.class 檔案。

來看下面範例 11.4 瞭解如何使用定義好的列舉型態。

範例 11.4 EnumDemo.java

```
public class EnumDemo {
    public static void main(String[] args) {
        doAction(Action.TURN_RIGHT);
    }

    public static void doAction(Action action) {
        switch(action) {
            case TURN_LEFT:
                System.out.println("向左轉");
                break;
            case TURN_RIGHT:
                System.out.println("向右轉");
                break;
        }
    }
}
```

```

        case SHOOT:
            System.out.println("射擊");
            break;
    }
}

```

執行結果：

```
向右轉
```

除了讓您少打一些字之外，這個範例好像沒有什麼特別的，但注意到 doAction() 參數列的型態是 Action，如果您對 doAction() 方法輸入其它型態的引數，編譯器會回報錯誤，因為 doAction() 所接受的引數必須是 Action 列舉型態。使用列舉型態還可以作到更進一步的檢驗，如果您在 "switch" 中加入了不屬於 Action 中列舉的值，編譯器也會回報錯誤，例如：

```

...
public static void doAction(Action action) {
    switch(action) {
        case TURN_LEFT:
            System.out.println("向左轉");
            break;
        case TURN_RIGHT:
            System.out.println("向右轉");
            break;
        case SHOOT:
            System.out.println("射擊");
            break;
        case STOP: // Action中沒有列舉這個值
            System.out.println("停止");
            break;
    }
}
...

```

在編譯時編譯器會替您作檢查，若檢查出不屬於 Action 中的列舉值，會顯示以下的錯誤：

```
unqualified enumeration constant name required
case STOP:
^
```

您可以在一個獨立的檔案中宣告列舉值，或是在某個類別中宣告列舉成員，例如範例 11.5 將 Action 列舉型態宣告於 EnumDemo2 類別中。

範例 11.5 EnumDemo2.java

```

public class EnumDemo2 {
    private enum InnerAction {TURN_LEFT, TURN_RIGHT, SHOOT};

    public static void main(String[] args) {
        doAction(InnerAction.TURN_RIGHT);
    }

    public static void doAction(InnerAction action) {
        switch(action) {
            case TURN_LEFT:
                System.out.println("向左轉");
                break;
            case TURN_RIGHT:
                System.out.println("向右轉");
                break;
            case SHOOT:
                System.out.println("射擊");
                break;
        }
    }
}

```

```
    }
}
}
```

執行結果：

```
向右轉
```

由於列舉型態本質上還是個類別，所以範例11.5的列舉宣告方式有些像在宣告「內部類別」（Inner class），在您編譯完 EnumDemo2.java 檔案後，除了 EnumDemo2.class 之外，您會有一些額外的 .class 檔案產生，在這個例子中就是 EnumDemo2\$InnerAction.class 與 EnumDemo2\$1.class，看到這兩個檔案，您就應該瞭解實際上編譯器產生了「成員內部類別」以及「匿名內部類別」（第 9 章說明過內部類別）。

11.2 定義列舉型態

就簡單的應用而言，上一個小節介紹的列舉型態入門，就比舊版本的常數設定方式多了編譯時期型態檢查的好處，然而列舉型態的功能還不止這些，這個小節中再介紹更多列舉型態的定義方式，您可以將這個小節介紹的內容當作另一種定義類別的方式，如此可以幫助您理解如何定義列舉型態。

11.2.1 深入列舉型態

定義列舉型態時其實就是在定義一個類別，只不過很多細節由編譯器幫您補齊了，所以某些程度上 "enum" 關鍵字的作用就像是 "class" 或 "interface"。

當您使用 "enum" 定義列舉型態時，實際上您所定義出來的型態是繼承自 `java.lang.Enum` 類別，而每個被列舉的成員其實就是您定義的列舉型態的一個實例，它們都被預設為 "final"，所以您無法改變常數名稱所設定的值，它們也是 "public" 且 "static" 的成員，所以您可以透過類別名稱直接使用它們。

舉個實際的例子，範例 11.3 定義了 `Action` 列舉型態，當中定義的 `TURN_LEFT`、`TURN_RIGHT`、`SHOOT` 都是 `Action` 的一個物件實例，因為是物件，所以物件上自然有一些方法可以呼叫使用，例如從 `Object` 繼承下來的 `toString()` 方法被重新定義了，可以讓您直接取得列舉值的字串描述；`values()` 方法可以讓您取得所有的列舉成員實例，並以陣列方式傳回，您可以使用這兩個方法來簡單的將 `Action`（要使用範例 11.3）的列舉成員顯示出來。

範例 11.6 ShowEnum.java

```
public class ShowEnum {
    public static void main(String[] args) {
        for(Action action: Action.values()) {
            System.out.println(action.toString());
        }
    }
}
```

基本上 `println()` 會自動呼叫物件 `toString()`，所以不寫 `toString()` 實際也是可以的，執行結果如下：

```
TURN_LEFT
TURN_RIGHT
SHOOT
```

由於每一個列舉的成員都是一個物件實例，所以您可以使用 "==" 或是 `equals()` 方法來比較列舉物件，"==" 會比較您提供的列舉物件是不是同一個物件，而 `equals()` 則是實質的比較兩個列舉物件的內容是否相等，使用 `equals()` 時預設會根據列舉物件的字串值來比較。

靜態 `valueOf()` 方法可以讓您將指定的字串嘗試轉換為列舉實例，您可以使用 `compareTo()` 方法來比較兩個列舉物件在列舉時的順序，範例是這兩個方法的實際例子。

範例 11.7 EnumCompareTo.java

```
public class EnumCompareTo {
    public static void main(String[] args) {
        compareToAction(Action.valueOf(args[0]));
    }

    public static void compareToAction(Action inputAction) {
        System.out.println("輸入：" + inputAction);
        for(Action action: Action.values()) {
            System.out.println(action.compareTo(inputAction));
```

```

        }
    }
}

```

`compareTo()` 如果傳回正值，表示設定為引數的列舉物件（`inputAction`）其順序在比較的列舉物件（`action`）之前，負值表示在之後，而0則表示兩個互比列舉值的位置是相同的，執行結果如下：

```

java EnumCompareTo SHOOT
輸入：SHOOT
-2
-1
0

```

對於每一個列舉成員，您可以使用 `ordinal()` 方法，依列舉順序得到位置索引，預設以 0 開始，範例 11.8 是個簡單示範。

範例 11.8 EnumIndex.java

```

public class EnumIndex {
    public static void main(String[] args) {
        for(Action action : Action.values()) {
            System.out.printf("%d %s%n", action.ordinal(), action);
        }
    }
}

```

執行結果：

```

0 TURN_LEFT
1 TURN_RIGHT
2 SHOOT

```

11.2.2 列舉上的方法

定義列舉型態基本上就像是在定義類別，定義列舉型態時您也可以定義方法，例如，您也許會想要為列舉值加上一些描述，而不是使用預設的 `toString()` 返回值來描述列舉值，如範例 11.9 所示。

範例 11.9 DetailAction.java

```

public enum DetailAction {
    TURN_LEFT, TURN_RIGHT, SHOOT;

    public String getDescription() {
        switch(this.ordinal()) {
            case 0:
                return "向左轉";
            case 1:
                return "向右轉";
            case 2:
                return "射擊";
            default:
                return null;
        }
    }
}

```

您可以使用範例 11.10 來測試一下所定義的方法是否有用。

範例 11.10 DetailActionDemo.java

```
public class DetailActionDemo {
    public static void main(String[] args) {
        for(DetailAction action : DetailAction.values()) {
            System.out.printf("%s : %s%n",
                action, action.getDescription());
        }
    }
}
```

執行結果：

```
TURN_LEFT : 向左轉
TURN_RIGHT : 向右轉
SHOOT : 射擊
```

列舉型態既然是類別，那麼您可以為它加上建構方法（Constructor）嗎？答案是可以的，但是不得為公開的（public）建構方法，這是為了避免粗心的程式設計人員直接對列舉型態實例化，一個不公開的建構方法可以作什麼？來看看下面範例 11.11 的實作。

範例 11.11 DetailAction2.java

```
public enum DetailAction2 {
    TURN_LEFT("向左轉"), TURN_RIGHT("向右轉"), SHOOT("射擊");

    private String description;

    // 不公開的建構方法
    private DetailAction2(String description) {
        this.description = description;
    }

    public String getDescription() {
        return description;
    }
}
```

在列舉 TURN_LEFT、TURN_RIGHT、SHOOT 成員時，您可以一併指定文字描述，這個描述會在建構列舉物件時使用，範例 11.11 中您將之設定給私用成員 description，在使用 getDescription() 時將之返回，您可以使用範例 11.10 加以修改（將 DetailAction 改為 DetailAction2），可以得到相同的顯示結果。

在定義列舉值時也可以一併實作介面（Interface），例如先來定義一個介面。

範例 11.12 IDescription.java

```
public interface IDescription {
    public String getDescription();
}
```

您可以使用這個介面規定每個實作該介面的列舉，都必須傳回一個描述列舉值的字串，如範例 11.13 所示。

範例 11.13 DetailAction3.java

```
public enum DetailAction3 implements IDescription {
    TURN_LEFT("向左轉"), TURN_RIGHT("向右轉"), SHOOT("射擊");
```

```

private String description;

// 不公開的建構方法
private DetailAction3(String description) {
    this.description = description;
}

public String getDescription() {
    return description;
}
}

```

良葛格的話匣子 非公開的建構方法最常見的例子就是「Singleton 模式」的應用，當某個類別只能有一個實例時，可由類別維護唯一的實例，這時可以將建構方法設定為私用（private），取用此類別的開發人員就不能自行新增多個實例了，可以看看我網站上有關 Singleton 模式的介紹：

- <http://openhome.cc/Gossip/DesignPattern/>

11.2.3 因值而異的類實作 (Value-Specific Class Bodies)

因值而異的類實作？原文為 Value-Specific Class Bodies，其實這個功能簡單的說，實作時像是在使用「匿名內部類別」（Anonymous inner class）（9.1.2 有介紹）來實現「Command 模式」，它讓您可以為每個列舉值定義各自的類本體與方法（Method）實作。

先來看看其中一種實現的方式，這邊要使用範例 11.12 的 IDescription 介面，您希望每個列舉的實例實作自己的 getDescription() 方法（而不是像範例 11.13 所介紹的，在定義列舉時實作一個統一的 getDescription() 方法），如範例 11.14 所示。

範例 11.14 MoreAction.java

```

public enum MoreAction implements IDescription {
    TURN_LEFT {
        // 實作介面上的方法
        public String getDescription() {
            return "向左轉";
        }
    }, // 記得這邊的列舉值分隔使用 ,

    TURN_RIGHT {
        // 實作介面上的方法
        public String getDescription() {
            return "向右轉";
        }
    }, // 記得這邊的列舉值分隔使用 ,

    SHOOT {
        // 實作介面上的方法
        public String getDescription() {
            return "射擊";
        }
    }; // 記得這邊的列舉值結束使用 ;
}

```

每個列舉成員的 '{' 與 '}' 之間是類本體，您還可以在當中如同定義類別一樣的宣告資料成員或實作方法。TURN_LEFT、TURN_RIGHT 與 SHOOT 三個 MoreAction 的列舉實例各自在本體（Body），也就是 '{' 與 '}' 之間實作了自己的 getDescription() 方法，而不是像範例 11.13 中統一實作在 DetailAction3 中，使用範例 11.15 作個測試。

範例 11.15 MoreActionDemo.java

```

public class MoreActionDemo {
    public static void main(String[] args) {
}

```

```

        for(MoreAction action : MoreAction.values()) {
            System.out.printf("%s : %s%n",
                action, action.getDescription());
        }
    }
}

```

這個例子是將「因值而異的類實作」用在返回列舉值描述上，您可以依相同的方式，為每個列舉值加上一些各自的方法實作，而呼叫的介面是統一的，執行結果會顯示各自的列舉描述：

```

TURN_LEFT : 向左轉
TURN_RIGHT : 向右轉
SHOOT : 射擊

```

您也可以運用抽象方法來改寫範例 11.14，如範例 11.16 所示。

範例 11.16 MoreAction2.java

```

public enum MoreAction2 {
    TURN_LEFT {
        // 實作抽象方法
        public String getDescription() {
            return "向左轉";
        }
    }, // 記得這邊的列舉值分隔使用 ,

    TURN_RIGHT {
        // 實作抽象方法
        public String getDescription() {
            return "向右轉";
        }
    }, // 記得這邊的列舉值分隔使用 ,

    SHOOT {
        // 實作抽象方法
        public String getDescription() {
            return "射擊";
        }
    }; // 記得這邊的列舉值結束使用 ;

    // 宣告個抽象方法
    public abstract String getDescription();
}

```

MoreAction2 與 MoreAction 不同的地方在於 MoreAction2 是實作抽象方法，您可以改寫一些範例 11.15（將 MoreAction 改為 MoreAction2），而執行結果是一樣的；基本上定義介面方法或抽象方法，是為了知道物件的操作介面，這樣您才能去操作這個物件。

11.3 接下來的主題

每一個章節的內容由淺至深，初學者該掌握的深度要到哪呢？在這個章節中，對於初學者我建議至少掌握以下幾點內容：

- 知道如何使用類別或介面定義以管理常數
- 會使用 "enum" 來取代常數列舉
- 知道列舉型態實際上就是在定義一個類別

下一個章節要介紹的也是 J2SE 5.0 的新功能：泛型（Generics）。除了可以讓您少寫幾個類別的程式碼之外，泛型的目的還在讓您定義「安全的」泛型類別（Generics class），事實上 J2SE 5.0 前就用 Object 解決了泛型類別的部份需求，J2SE 5.0 之後再解決的是型態安全問題。

第 12 章 泛型

在 J2SE 5.0 中新增了「泛型」（Generics）功能，而且許多 API 都根據這個新功能重新改寫了，例如 List、Map、Set 等相關類別，雖然即使不瞭解泛型的新功能，也可以照 J2SE 1.4 或舊版本的語法來使用這些類別，但編譯時會出現一些惱人的警訊（Warnings）。

泛型解決的不只是讓您少寫幾個類別的程式碼，還在於讓您定義「安全的」泛型類別（Generics class），泛型提供編譯時期檢查，您不會因為將物件置入某個容器（Container）而失去其型態，瞭解這一個章節的內容後，對於使用其它 API 有關泛型的功能而出現的警訊，您就知道其原因何在了。

12.1 泛型入門

J2SE 5.0 提供的泛型，目的在讓您定義「安全的」泛型類別（Generics class），事實上 J2SE 5.0 前 Object 解決泛型類別的部份需求，J2SE 5.0 之後再解決的是型態安全問題，這個小節會先介紹沒有泛型功能前的設計方法，再來看看幾個 J2SE 5.0 泛型的定義方式，從中瞭解使用泛型的好處。

12.1.1 沒有泛型之前

考慮您要設計下面的 BooleanFoo 與 IntegerFoo 兩個類別，這是兩個很無聊的類別，但足以說明需求。

範例 12.1 BooleanFoo.java

```
public class BooleanFoo {
    private Boolean foo;

    public void setFoo(Boolean foo) {
        this.foo = foo;
    }

    public Boolean getFoo() {
        return foo;
    }
}
```

範例 12.2 IntegerFoo.java

```
public class IntegerFoo {
    private Integer foo;

    public void setFoo(Integer foo) {
        this.foo = foo;
    }

    public Integer getFoo() {
        return foo;
    }
}
```

觀察範例 12.1 與 12.2 兩個類別，其中除了宣告成員的型態、參數列的型態與方法返回值的型態之外，剩下的程式碼完全相同，或許有點小聰明的程式設計人員會將第一個類的內容複製至另一個檔案中，然後用編輯器「取代」功能一次取代所有的型態名稱（即將 Boolean 取代為 Integer）。

雖然是有些小聰明，但如果類別中的邏輯要修改，您就需要修改兩個檔案，泛型（Generics）的需求就在此產生，當您定義類別時，發現到好幾個類別的邏輯其實都相同，就只是當中所涉及的型態不一樣時，使用複製、貼上、取代的功能來撰寫程式，只是讓您增加不必要的檔案管理困擾。

由於 Java 中所有的類別最上層都繼承自 Object 類別，您可以定義如範例 12.3 的類別來取代範例 12.1 與 12.2 的類別。

範例 12.3 ObjectFoo.java

```
public class ObjectFoo {
    private Object foo;

    public void setFoo(Object foo) {
        this.foo = foo;
    }
}
```

```

public Object getFoo() {
    return foo;
}
}

```

由於 Java 中所有定義的類別，都以 Object 為最上層的父類別，所以用它來實現泛型（Generics）功能是一個不錯的考量，在 J2SE 1.4 或之前版本上，大部份的開發人員會這麼作，您只要撰寫如範例 12.3 的類別，然後可以如下的使用它：

```

ObjectFoo foo1 = new ObjectFoo();
ObjectFoo foo2 = new ObjectFoo();

foo1.setFoo(new Boolean(true));
// 記得轉換操作型態
Boolean b = (Boolean) foo1.getFoo();

foo2.setFoo(new Integer(10));
// 記得轉換操作型態
Integer i = (Integer) foo2.getFoo();

```

看來還不錯，但是設定至 foo1 或 foo2 的 Integer 或 Boolean 實例會失去其型態資訊，從 getFoo() 傳回的是 Object 型態的實例，您必須轉換它的操作型態，問題出在這邊，粗心的程式設計人員往往會忘了要作這個動作，或者是轉換型態時用錯了型態（像是該用 Boolean 却用了 Integer），例如：

```

ObjectFoo foo1 = new ObjectFoo();
foo1.setFoo(new Boolean(true));
String s = (String) foo1.getFoo();

```

由於語法上並沒有錯誤，所以編譯器檢查不出上面的程式有錯誤，真正的錯誤要在執行時期才會發生，這時惱人的 ClassCastException 就會出來搞怪，在使用 Object 設計泛型程式時，程式人員要再細心一些，例如在 J2SE 1.4 或舊版本上，所有存入 List、Map、Set 容器中的實例都會失去其型態資訊，要從這些容器中取回物件並加以操作的話，就得記住取回的物件是什麼型態。

12.1.2 定義泛型類別

當您定義類別時，發現到好幾個類別的邏輯其實都相同，就只是當中所涉及的型態不一樣時，使用複製、貼上、取代的功能來撰寫程式，只會讓您增加不必要的檔案管理困擾。

由於 Java 中所有定義的類別，都以 Object 為最上層的父類別，所以在 J2SE 5.0 之前，Java 程式設計人員可以使用 Object 定義類別以解決以上的需求，為了讓定義出來的類別可以更加通用（Generic），傳入的值或傳回的實例都是以 Object 型態為主，當您要取出這些實例來使用時，必須記得將之轉換為原來的類型或適當的介面，如此才可以操作物件上的方法。

然而使用 Object 來撰寫泛型類別（Generic Class）留下了一些問題，因為您必須要轉換型態或介面，粗心的程式設計人員往往會忘了要作這個動作，或者是轉換型態或介面時用錯了型態或介面（像是該用 Boolean 却用了 Integer），但由於語法上是可以的，所以編譯器檢查不出錯誤，因而執行時期就會發生 ClassCastException。

在 J2SE 5.0 之後，提出了針對泛型（Generics）設計的解決方案，要定義一個簡單的泛型類別是簡單的，直接來看範例 12.4 如何取代範例 12.3 的類別定義。

範例 12.4 GenericFoo.java

```

public class GenericFoo<T> {
    private T foo;

    public void setFoo(T foo) {
        this.foo = foo;
    }
}

```

```

public T getFoo() {
    return foo;
}
}

```

在範例 12.4 中，使用 `<T>` 用來宣告一個型態持有者（Holder）名稱 `T`，之後您可以用 `T` 這個名稱作為型態代表來宣告成員、參數或返回值型態，然後您可以如範例 12.5 來使用這個類別。

範例 12.5 GenericFooDemo.java

```

public class GenericFooDemo {
    public static void main(String[] args) {
        GenericFoo<Boolean> foo1 = new GenericFoo<Boolean>();
        GenericFoo<Integer> foo2 = new GenericFoo<Integer>();

        foo1.setFoo(new Boolean(true));
        Boolean b = foo1.getFoo(); // 不需要再轉換型態
        System.out.println(b);

        foo2.setFoo(new Integer(10));
        Integer i = foo2.getFoo(); // 不需要再轉換型態
        System.out.println(i);
    }
}

```

與單純使用 `Object` 壓告型態所不同的地方在於，使用泛型所定義的類別在宣告及配置物件時，您可以使用角括號一併指定泛型類別型態持有者 `T` 真正的型態，而型態或介面轉換就不再需要了，`getFoo()` 所設定的引數或傳回的型態，就是您在宣告及配置物件時在 `<>` 之間所指定的型態，您所定義出來的泛型類別在使用時多了一層安全性，可以省去惱人的 `ClassCastException` 發生，編譯器可以幫您作第一層防線，例如下面的程式會被檢查出錯誤：

```

GenericFoo<Boolean> foo1 = new GenericFoo<Boolean>();
foo1.setFoo(new Boolean(true));
Integer i = foo1.getFoo(); // 傳回的是Boolean型態

```

`foo1` 使用 `getFoo()` 方法傳回的是 `Boolean` 型態的實例，若您要將這個實例指定給 `Integer` 型態的變數，顯然在語法上不合，編譯器這時檢查出錯誤：

```

GenericFooDemo.java:7: incompatible types
found : java.lang.Boolean
required: java.lang.Integer
Integer i = foo1.getFoo();

```

如果使用泛型類別，但宣告及配置物件時不一併指定型態呢？那麼預設會使用 `Object` 型態，不過您就要自己轉換物件的介面型態了，例如 `GenericFoo` 可以這麼宣告與使用：

```

GenericFoo foo3 = new GenericFoo();
foo3.setFoo(new Boolean(false));

```

但編譯時編譯器會提出警訊，告訴您這可能是不安全的操作：

```

Note: GenericFooDemo.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

```

回過頭來看看下面的宣告：

```
GenericFoo<Boolean> foo1 = new GenericFoo<Boolean>();
GenericFoo<Integer> foo2 = new GenericFoo<Integer>();
```

`GenericFoo<Boolean>` 壓告的 `foo1` 與 `GenericFoo<Integer>` 壓告的 `foo2` 是相同的類型嗎？答案是否定的！基本上 `foo1` 與 `foo2` 是兩個不同的類型，`foo1` 是 `GenericFoo<Boolean>` 類型，而 `foo2` 是 `GenericFoo<Integer>` 類型，所以您不可以將 `foo1` 所參考的實例指定給 `foo2`，或是將 `foo2` 所參考的實例指定給 `foo1`，要不然編譯器會回報以下錯誤：

```
incompatible types
found : GenericFoo<java.lang.Integer>
required: GenericFoo<java.lang.Boolean>
foo1 = foo2;
```

良葛格的話匣子 自訂義泛型類別時，型態持有者名稱可以使用 `T` (Type)，如果是容器的元素可以使用 `E` (Element)，鍵值匹配的話使用 `K` (Key) 與 `V` (Value)，Annotation 的話可以用 `A`，可以參考 J2SE 5.0 API 文件說明上的命名方式。

12.1.3 幾個定義泛型的例子

您可以在定義泛型類別時，宣告多個類型持有者，像範例 12.6 的類別上宣告了兩個型態持有者 `T1` 與 `T2`。

範例 12.6 GenericFoo2.java

```
public class GenericFoo2<T1, T2> {
    private T1 foo1;
    private T2 foo2;

    public void setFoo1(T1 foo1) {
        this.foo1 = foo1;
    }

    public T1 getFoo1() {
        return foo1;
    }

    public void setFoo2(T2 foo2) {
        this.foo2 = foo2;
    }

    public T2 getFoo2() {
        return foo2;
    }
}
```

您可以如下使用 `GenericFoo2` 類別，分別以 `Integer` 與 `Boolean` 設定 `T1` 與 `T2` 的真正型態：

```
GenericFoo<Integer, Boolean> foo =
    new GenericFoo<Integer, Boolean>();
```

泛型可以用於宣告陣列型態，範例 12.7 是個簡單示範。

範例 12.7 GenericFoo3.java

```
public class GenericFoo3<T> {
    private T[] fooArray;

    public void setFooArray(T[] fooArray) {
        this.fooArray = fooArray;
    }
}
```

```

    }

    public T[] getFooArray() {
        return fooArray;
    }
}

```

您可以像下面的方式來使用範例 12.7 所定義的類別。

```

String[] strs = {"caterpillar", "momor", "bush"};
GenericFoo3<String> foo = new GenericFoo3<String>();
foo.setFooArray(strs);
strs = foo.getFooArray();

```

注意您可以使用泛型機制來宣告一個陣列，例如下面這樣是可行的：

```

public class GenericFoo<T> {
    private T[] fooArray;
    // ...
}

```

但是您不可以使用泛型來建立陣列的實例，例如以下是不可行的：

```

public class GenericFoo<T> {
    private T[] fooArray = new T[10]; // 不可以使用泛型建立陣列實例
    // ...
}

```

如果您已經定義了一個泛型類別，想要用這個類別在另一個泛型類別中宣告成員的話要如何作？舉個實例，假設您已經定義了範例 12.4 的類別，現在想要設計一個新的類別，當中包括了範例 12.4 的類別實例作為其成員，您可以如範例 12.8 的方式設計。

範例 12.8 WrapperFoo.java

```

public class WrapperFoo<T> {
    private GenericFoo<T> foo;

    public void setFoo(GenericFoo<T> foo) {
        this.foo = foo;
    }

    public GenericFoo<T> getFoo() {
        return foo;
    }
}

```

這麼一來，您就可以保留型態持有者T的功能，一個使用的例子如下：

```

GenericFoo<Integer> foo = new GenericFoo<Integer>();
foo.setFoo(new Integer(10));
WrapperFoo<Integer> wrapper = new WrapperFoo<Integer>();
wrapper.setFoo(foo);

```

12.2 泛型進階語法

泛型的語法元素其實是很基本的，只不過將這種語法遞迴擴展之後，可以撰寫出相當複雜的泛型定義，然而無論再怎麼複雜的寫法，基本語法元素大致不離：限制泛型可用類型、使用型態通配字元（Wildcard）、以及泛型的擴充與繼承這幾個語法。

12.2.1 限制泛型可用類型

在定義泛型類別時，預設您可以使用任何的型態來實例化泛型類別中的型態持有者，但假設您想要限制使用泛型類別時，只能用某個特定型態或其子類別來實例化型態持有者的話呢？

您可以在定義型態持有者時，一併使用 "extends" 指定這個型態持有者實例化時，實例化的對象必須是擴充自某個類型或實作某介面，舉範例 12.9 來說。

範例 12.9 ListGenericFoo.java

```
import java.util.List;

public class ListGenericFoo<T extends List> {
    private T[] fooArray;

    public void setFooArray(T[] fooArray) {
        this.fooArray = fooArray;
    }

    public T[] getFooArray() {
        return fooArray;
    }
}
```

ListGenericFoo 在宣告類型持有者時，一併指定這個持有者實例化的對象，必須是實作 java.util.List 介面（interface）的類別，在限定持有者時，無論是要限定的對象是介面或類別，都是使用 "extends" 關鍵字，範例中您使用 "extends" 限定型態持有者實例化的對象，必須是實作 List 介面的類別，像 java.util.LinkedList 與 java.util.ArrayList 就實作了 List 介面（第 13 章就會介紹），例如下面的程式片段是合法的使用方式：

```
ListGenericFoo<LinkedList> foo1 =
    new ListGenericFoo<LinkedList>();
ListGenericFoo<ArrayList> foo2 =
    new ListGenericFoo<ArrayList>();
```

但如果不是實作 List 的類別，編譯時就會發生錯誤，例如下面的程式片段通不過編譯：

```
ListGenericFoo<HashMap> foo3 =
    new ListGenericFoo<HashMap>();
```

因為 java.util.HashMap 並沒有實作 List 介面（事實上 HashMap 實作了 Map 介面），編譯器會在編譯時期就檢查出這個錯誤：

```
type parameter java.util.HashMap is not within its bound
ListGenericFoo<HashMap> foo3 = new ListGenericFoo<HashMap>();
```

HashMap 並沒有實作 List 介面，所以無法作為實例化型態持有者的對象，事實上，當您沒有使用 "extends" 關鍵字限定型態泛型進階語法

持有者時，預設是 Object 下的所有子類別都可以實例化型態持有者，也就是說在您定義泛型類別時如果只寫以下的話：

```
public class GenericFoo<T> {
    //...
}
```

其實就相當於以下的定義方式：

```
public class GenericFoo<T extends Object> {
    //...
}
```

由於 Java 中所有的實例都繼承自 Object 類別，所以定義時若只寫 `<T>` 就表示，所有類型的物件都可以實例化您所定義的泛型類別。

良葛格的話匣子 實際上由於 List、Map、Set 與實作這些介面的相關類別，都已經用新的泛型功能重新改寫過了，實際撰寫時會更複雜一些，例如實際上您還可以再細部定義範例 12.9 的 ListGenericFoo：

```
import java.util.List;
public class ListGenericFoo<T extends List<String>> {
    private T[] fooArray;
    public void setFooArray(T[] fooArray) {
        this.fooArray = fooArray;
    }
    public T[] getFooArray() {
        return fooArray;
    }
}
```

這麼定義之後，您就只能使用 `ArrayList<String>` 來實例化 ListGenericFoo 了，例如：

```
ListGenericFoo<ArrayList<String>> foo =
    new ListGenericFoo<ArrayList<String>>();
```

下一個章節會說明 List、Map、Set 等的使用，雖然展開後的程式似乎很複雜，但實際上還是這個章節所介紹泛型語法的延伸，為了說明方便，在這個章節中，請先忽略 List、Map、Set 上的泛型定義，先當它是個介面就好了。

12.2.2 型態通配字元（Wildcard）

仍然以範例 12.4 所定義的 GenericFoo 來進行說明，假設您使用 GenericFoo 類別來如下宣告名稱：

```
GenericFoo<Integer> foo1 = null;
GenericFoo<Boolean> foo2 = null;
```

那麼名稱 foo1 就只能參考 `GenericFoo<Integer>` 類型的實例，而名稱 foo2 只能參考 `GenericFoo<Boolean>` 類型的實例，也就是說下面的方式是可行的：

```
foo1 = new GenericFoo<Integer>();
foo2 = new GenericFoo<Boolean>();
```

現在您有這麼一個需求，您希望有一個參考名稱 foo 可以如下接受所指定的實例：

```
foo = new GenericFoo<ArrayList>();
foo = new GenericFoo<LinkedList>();
```

簡單的說，您想要有一個 foo 名稱可以參考的對象，其型態持有者實例化的對象是實作 List 介面的類別或其子類別，要宣告這麼一個參考名稱，您可以使用 '?' 「通配字元」（Wildcard），'?' 代表未知型態，並使用 "extends" 關鍵字來作限定，例如：

```
GenericFoo<? extends List> foo = null;
foo = new GenericFoo<ArrayList>();
.....
foo = new GenericFoo<LinkedList>();
....
```

<? extends List> 表示型態未知，只知會是實作 List 介面的類別，所以如果型態持有者實例化的對象不是實作 List 介面的類別，則編譯器會回報錯誤，例如以下這行無法通過編譯：

```
GenericFoo<? extends List> foo = new GenericFoo<HashMap>();
```

因為 HashMap 沒有實作 List 介面，所以建立的 GenericFoo<HashMap> 實例不能指定給 foo 名稱來參考，編譯器會回報以下的錯誤：

```
incompatible types
found : GenericFoo<java.util.HashMap>
required: GenericFoo<? extends java.util.List>
GenericFoo<? extends List> foo = new GenericFoo<HashMap>();
```

使用 '?' 來作限定有時是很有用的，例如若您想要自訂一個 showFoo() 方法，方法的內容實作是針對 String 或其子類的實例而制定的，例如：

```
public void showFoo(GenericFoo foo) {
    // 針對String或其子類而制定的內容
}
```

如果只作以上的宣告，那麼像 GenericFoo<Integer>、 GenericFoo<Boolean> 等型態都可以傳入至方法中，如果您不希望任何的型態都可以傳入 showFoo() 方法中，您可以使用以下的方式來限定：

```
public void showFoo(GenericFoo<? extends String> foo) {
    // 針對String或其子類而制定的內容，例如下面這行
    System.out.println(foo.getFoo());
}
```

這麼一來，如果有粗心的程式設計人員傳入了您不想要的型態，例如 GenericFoo<Boolean> 型態的實例，則編譯器都會告訴它這是不可行的，在宣告名稱時如果指定了 <?> 而不使用 "extends"，則預設是允許 Object 及其下的子類，也就是所有的 Java 物件了，那為什麼不直接使用 GenericFoo 告訴就好了，何必要用 GenericFoo<?> 來宣告？使用通配字元有點要注意的是，透過使用通配字元宣告的名稱所參考的物件，您沒辦法再對它加入新的資訊，您只能取得它當中的資訊或是移除當中的資訊，例如：

```
GenericFoo<String> foo = new GenericFoo<String>();
foo.setFoo("caterpillar");

GenericFoo<?> immutableFoo = foo;
// 可以取得資訊
```

```

System.out.println(immutableFoo.getFoo());

// 可透過immutableFoo來移去foo所參考實例內的資訊
immutableFoo.setFoo(null);

// 不可透過immutableFoo來設定新的資訊給foo所參考的實例
// 所以下面這行無法通過編譯
// immutableFoo.setFoo("良葛格");

```

所以使用 `<?>` 或是 `<? extends SomeClass>` 的宣告方式，意味著您只能透過該名稱來取得所參考實例的資訊，或者是移除某些資訊，但不能增加它的資訊，理由很簡單，因為您不知道 `<?>` 或是 `<? extends SomeClass>` 宣告的參考名稱，實際上參考的物件，當中確實儲存的是什麼類型的資訊，基於泛型的設計理念，當然也就沒有理由能加入新的資訊了，因為若能加入，被加入的物件同樣也會有失去型態資訊的問題。

除了可以向下限制，您也可以向上限制，只要使用 "super" 關鍵字，例如：

```
GenericFoo<? super StringBuilder> foo = null;
```

如此，`foo` 就只接受 `StringBuilder` 及其上層的父類型態，也就是只能接受 `GenericFoo<StringBuilder>` 與 `GenericFoo<Object>` 的實例。

12.2.3 擴充泛型類別、實作泛型介面

您可以擴充一個泛型類別，保留其型態持有者，並新增自己的型態持有者，例如範例12.10先寫一個父類別。

範例 12.10 GenericFoo4.java

```

public class GenericFoo4<T1, T2> {
    private T1 foo1;
    private T2 foo2;

    public void setFoo1(T1 foo1) {
        this.foo1 = foo1;
    }

    public T1 getFoo1() {
        return foo1;
    }

    public void setFoo2(T2 foo2) {
        this.foo2 = foo2;
    }

    public T2 getFoo2() {
        return foo2;
    }
}

```

再如範例 12.11 寫一個子類別擴充範例 12.10 的父類別。

範例 12.11 SubGenericFoo4.java

```

public class SubGenericFoo4<T1, T2, T3>
    extends GenericFoo4<T1, T2> {
private T3 foo3;

    public void setFoo3(T3 foo3) {
        this.foo3 = foo3;
    }

    public T3 getFoo3() {

```

```

        return foo3;
    }
}

```

如果決定要保留型態持有者，則父類別上宣告的型態持有者數目在繼承下來時必須寫齊全，也就是說在範例 12.11 中，父類上 GenericFoo4 上出現的 T1 與 T2 在 SubGenericFoo4 中都要出現，如果不保留型態持有者，則繼承下來的 T1 與 T2 自動變為 Object，建議是父類別的型態持有者都要保留。介面實作也是類似，例如先如範例 12.12 定義一個介面。

範例 12.12 IFoo.java

```

public interface IFoo<T1, T2> {
    public void setFoo1(T1 foo1);
    public void setFoo2(T2 foo2);
    public T1 getFoo1();
    public T2 getFoo2();
}

```

您可以如範例 12.13 的方式實作 IFoo 介面，實作時保留所有的型態持有者。

範例 12.13 ConcreteFoo.java

```

public class ConcreteFoo<T1, T2> implements IFoo<T1, T2> {
    private T1 foo1;
    private T2 foo2;

    public void setFoo1(T1 foo1) {
        this.foo1 = foo1;
    }

    public T1 getFoo1() {
        return foo1;
    }

    public void setFoo2(T2 foo2) {
        this.foo2 = foo2;
    }

    public T2 getFoo2() {
        return foo2;
    }
}

```

良葛格的話匣子 有的人一看到角括號就開始頭痛，老實說我也是這些人當中的一個，Java 新增的泛型語法雖基本，但根據語法展開來的寫法卻可以寫的很複雜，我不建議使用泛型時將程式碼寫的太複雜，像是遞迴了 n 層角括號的程式碼，看來真的很令人頭痛，新的泛型功能有其好處，但撰寫程式時也要同時考慮可讀性，因為可讀性有時反而是開發程式時比較注重的。

12.3 接下來的主題

每一個章節的內容由淺至深，初學者該掌握的深度要到哪呢？在這個章節中，對於初學者我建議至少掌握以下幾點內容：

- 瞭解沒有泛型功能前，如何使用 Object 解決問題
- 會定義基本的泛型類別及宣告其實例（即 12.1.2 的內容）
- 會擴充泛型類別與實作泛型介面

到這個章節為止，Java 的語法大部份已經說明完畢了，接下來要進行的，是讓您開始熟悉 J2SE 中一些常用的 API 類別，熟悉這些 API 類別是學習 Java 的必要過程，首先要先瞭解的是相關的物件容器（Container），像是 List、Map、Set 等，幾乎在各種應用領域中，都會很常使用到這些物件容器。

第 13 章 物件容器（Container）

在程式運行的過程中，很多時候您需要將物件暫時儲存在一個容器中統一管理，之後需要時再將物件取出，要使用什麼樣的容器依設計需求而定，您可以使用循序有索引的串列（List）結構來儲存物件，或是使用不允許重複相同物件的集合（Set）結構，您也可以使用「鍵-值」（Key-Value）存取的Map。

儲存結構與資料結構的設計是息息相關的，使用 Java SE 的話，即使您實際上不瞭解 List、Set、Map 資料結構的設計方式，也可以直接取用實作 java.util.List、java.util.Set、java.util.Map 介面的相關類別，以得到使用相同資料結構的功能，在 J2SE 5.0 中，這些容器類別更使用了「泛型」（Generics）功能重新改寫，如此您就不用擔心物件儲存至容器就失去其型態資訊的問題。